KNITRO™ 5.0

User's Manual

KNITRO User's Manual

Version 5.0

Richard A. Waltz
Todd D. Plantenga
Ziena Optimization, Inc.
www.ziena.com

February 2006

©2004-2006 Ziena Optimization, Inc.

Contents

Contents **a**

1 Introduction **1**
1.1 Product Overview . . . 1
1.2 Algorithms Overview . . . 2
1.3 Contact and Support Information . . . 2

2 Installation **4**
2.1 Windows . . . 4
2.2 Linux/Unix . . . 5

3 Using KNITRO with the AMPL modeling language **8**

4 The KNITRO callable library **15**
4.1 KNITRO in a C application . . . 15
4.2 Examples of calling in C . . . 21
4.3 KNITRO in a C++ application . . . 26
4.4 KNITRO in a Java application . . . 26
4.5 KNITRO in a Fortran application . . . 26
4.6 Specifying the Jacobian and Hessian matrices in sparse form . . . 27

5 User options in KNITRO **30**
5.1 Description of KNITRO user options . . . 30
5.2 The KNITRO options file . . . 37
5.3 Setting options through function calls . . . 37

6 KNITRO termination test and optimality **39**

7 KNITRO output and solution information **41**
7.1 Understanding KNITRO output . . . 41
7.2 Accessing solution information . . . 44

8 Algorithm Options **45**
8.1 Automatic . . . 45
8.2 Interior/Direct . . . 45
8.3 Interior/CG . . . 45
8.4 Active . . . 45

9 Other KNITRO special features **46**
9.1 First derivative and gradient check options . . . 46
9.2 Second derivative options . . . 46
9.3 Feasible version . . . 47
9.4 Honor Bounds . . . 48
9.5 Crossover . . . 48
9.6 Multi-start . . . 49

9.7 Reverse communication mode for invoking KNITRO 49
9.8 Callback mode for invoking KNITRO 50

10 Special problem classes **52**
10.1 Linear programming problems (LPs) 52
10.2 Quadratic programming problems (QPs) 52
10.3 Systems of Nonlinear Equations 52
10.4 Least Squares Problems 52
10.5 Mathematical programs with equilibrium constraints (MPECs) 53
10.6 Global optimization 56

References **56**

Appendix A Solution Status Codes **58**

Appendix B Migrating to KNITRO 5.0 **60**

1 Introduction

This chapter gives an overview of the KNITRO optimization software package, and details concerning contact and support information.

1.1 Product Overview

KNITRO 5.0 is a software package for finding solutions of continuous, smooth optimization problems, with or without constraints. KNITRO is designed for finding local solutions but multi-start heuristics are provided for trying to locate the global solution. Although KNITRO is designed for solving large-scale general nonlinear problems, it is efficient at solving all of the following classes of smooth optimization problems:

- unconstrained,
- bound constrained,
- equality constrained, both linear and nonlinear,
- systems of nonlinear equations,
- least squares problems, both linear and nonlinear,
- linear programming problems (LPs),
- quadratic programming problems (QPs), both convex and nonconvex,
- mathematical programs with equilibrium constraints (MPECs),
- general nonlinear constrained problems (NLP), both convex and nonconvex.

The KNITRO package provides the following features:

- efficient and robust solution of small or large problems,
- derivative-free, 1st derivative and 2nd derivative options,
- options to remain feasible throughout the optimization or not,
- both interior-point (barrier) and active-set optimizers,
- both iterative and direct approaches for computing Newton-like steps,
- programmatic interfaces: C/C++, Fortran, Java, Microsoft Excel,
- modeling language interfaces: AMPL, GAMS, Mathematica, Matlab,
- reverse communication design for more user control over the optimization process and for easily embedding KNITRO within another piece of software.

1.2 Algorithms Overview

The problems solved by KNITRO have the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && \text{(1.1a)} \\ \text{subject to} \quad & h(x) = 0 && \text{(1.1b)} \\ & g(x) \leq 0, && \text{(1.1c)} \end{aligned}$$

where $x \in \mathbf{R}^n$. This formulation allows many types of constraints, including bounds on the variables. KNITRO assumes that the functions $f(x)$, $h(x)$ and $g(x)$ are smooth, although problems with derivative discontinuities can often be solved successfully.

KNITRO implements both state-of-the-art interior-point and active-set methods for solving nonlinear optimization problems. In the interior method (also known as a barrier method), the nonlinear programming problem is replaced by a series of barrier sub-problems controlled by a barrier parameter μ. The algorithm uses trust regions and a merit function to promote convergence. The algorithm performs one or more minimization steps on each barrier problem, then decreases the barrier parameter, and repeats the process until the original problem (1.1) has been solved to the desired accuracy.

KNITRO provides two procedures for computing the steps within the interior point approach. In the version known as `Interior/CG` each step is computed using a projected conjugate gradient iteration. This approach differs from most interior methods proposed in the literature in that it does not compute each step by solving a linear system involving the KKT (or primal-dual) matrix. Instead, it factors a projection matrix, and uses the conjugate gradient method, to approximately minimize a quadratic model of the barrier problem.

The second procedure for computing the steps, which we call `Interior/Direct`, always attempts to compute a new iterate by solving the primal-dual KKT matrix using direct linear algebra. In the case when this step cannot be guaranteed to be of good quality, or if negative curvature is detected, then the new iterate is computed by the `Interior/CG` procedure.

KNITRO also implements an active-set sequential linear-quadratic programming (SLQP) algorithm which we call `Active`. This method is similar in nature to a sequential quadratic programming method but uses linear programming sub-problems to estimate the active-set at each iteration. This active-set code may be preferable when a good initial point can be provided; for example, when solving a sequence of related problems.

We encourage the user to try all algorithmic options to determine which one is more suitable for the application at hand. For guidance on choosing the best algorithm see section 8.

For a detailed description of the algorithm implemented in `Interior/CG` see [4] and for the global convergence theory see [1]. The method implemented in `Interior/Direct` is described in [11]. The `Active` algorithm is described in [3] and the global convergence theory for this algorithm is in [2]. A summary of the algorithms and techniques implemented in the KNITRO software product is given in [6]. An important component of KNITRO is the HSL routine `MA27` [8] which is used to solve the linear systems arising at every iteration of the algorithm. In addition, the `Active` algorithm in KNITRO may make use of the COIN-OR Clp linear programming solver module. The version used in KNITRO may be downloaded from `http://www.ziena.com/clp.html`.

1.3 Contact and Support Information

KNITRO is licensed and supported by Ziena Optimization, Inc. (`http://www.ziena.com/`). General information regarding KNITRO can be found at the KNITRO website:

http://www.ziena.com/knitro.html

For technical support, contact your local distributor. If you purchased KNITRO directly from Ziena, you may send support questions or comments to

support-knitro@ziena.com

Questions regarding licensing information or other information about KNITRO can be sent to

info-knitro@ziena.com

2 Installation

Instructions for installing the KNITRO package on both Windows and Linux/UNIX platforms are described below. After installing, test KNITRO by compiling and running one of the examples. If you have AMPL, then refer to section 3 and test KNITRO as the solver for any smooth optimization model (an AMPL test model is provided with the KNITRO distribution).

2.1 Windows

The KNITRO 5.0 software package for Windows is delivered as a zipped file named knitro-5.0.0-z-WinMSVC.zip, or as a self-extracting executable named knitro-5.0.0-z-WinMSVC.exe. For the .zip file, double-click on it and extract all contents to a new folder. For the .exe file, double-click on it and follow the instructions. The self-extracting executable creates start menu shortcuts and an uninstall entry in Add/Remove Programs; otherwise, the two install methods are identical. The default installation location for KNITRO is

```
C:\Program Files\Ziena\
```

Unpacking will create a folder named knitro-5.0.0-z (or knitro-5.0.0-student for the student edition) containing the following files and subdirectories:

INSTALL.txt: A file containing installation instructions.

LICENSE_KNITRO.txt: A file containing the KNITRO license agreement.

README.txt: A file with instructions on how to get started using KNITRO.

KNITRO50-ReleaseNotes.txt: A file containing 5.0 release notes.

get_machine_ID.exe: An executable that identifies the machine ID, required for obtaining a Ziena license file.

doc: A folder containing KNITRO documentation, including this manual.

include: A folder containing the KNITRO header file knitro.h.

lib: A folder containing the KNITRO library and object files: knitro.lib, knitro_objlib.a, and knitro.dll.

examples: A folder containing examples of how to use the KNITRO API in different programming languages (C, Fortran, Java).

knitroampl: A folder containing files, instructions, and an example model for using KNITRO with AMPL.

To activate KNITRO for your computer you will need a valid Ziena license file. For a single machine license, execute get_machine_ID.exe, a program supplied with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Email the machine ID to `info@ziena.com` if purchased through Ziena. (If KNITRO was purchased through a distributor, then email the machine ID to your local distributor.) Ziena (or your local distributor) will then send a license file of the form `ziena_lic_name.txt`. The file name is arbitrary, so long as it begins with `"ziena_"`. The license file is made available to KNITRO in any one of the following ways:

- Put `ziena_lic_name.txt` in the folder `C:\Program Files\Ziena\`.
- Put `ziena_lic_name.txt` in the folder in which you run your KNITRO application.
- Put `ziena_lic_name.txt` anywhere and set the environment variable `ZIENA_LICENSE` to its location (e.g., `set ZIENA_LICENSE=C:\...\ziena_lic_name.txt`).

If you have difficulty installing the license file, then set the environment variable `ZIENA_LICENSE_DEBUG` and the license manager will display helpful information. Type `set ZIENA_LICENSE_DEBUG=1` and then try running KNITRO again.

Before beginning, view the INSTALL.txt, LICENSE_KNITRO.txt and README.txt files. To get started, build and test the programs provided inside the examples directory. Example problems are provided for the C, Fortran, and Java interfaces. We recommend understanding these examples and reading section 4 of this manual before proceeding with development of your own application interface. The knitroampl subdirectory contains instructions for using KNITRO with AMPL and an example of how to formulate a nonlinear optimization problem in AMPL.

The KNITRO 5.0 install locations on Windows are a departure from the 4.0 locations. To run KNITRO 4.0 and 5.0 on the same Windows machine, we recommend moving the 4.0 files to the new default location. At the very least, make sure the 4.0 and 5.0 executables and DLL file are not mixed together when compiling or running an application. We recommend uninstalling KNITRO 4.0 and reinstalling to a new folder

```
C:\Program Files\Ziena\knitro-4.0\
```

The effect of reinstalling should be to move the following KNITRO 4.0 files from their default locations to a new location:

`C:\Program Files\knitro-4.0\*`	to `C:\Program Files\Ziena\knitro-4.0\.`
`%SystemRoot%\knitro-ampl.exe`	to `C:\Program Files\Ziena\knitro-4.0\.`
`%SystemRoot%\system32\knitro.dll`	to `C:\Program Files\Ziena\knitro-4.0\lib\.`

KNITRO 4.0 and 5.0 require separate Ziena license files. Place both files in `C:\Program Files\Ziena\`.

2.2 Linux/Unix

The KNITRO 5.0 software package for Linux/Unix is delivered as a gzipped tar file named knitro-5.0.0-platformname.tar.gz. Save this file in a fresh subdirectory on your system. To unpack, type the commands

```
gunzip knitro-5.0.0-platformname.tar.gz
tar -xvf knitro-5.0.0-platformname.tar
```

This creates a directory named knitro-5.0.0-z (or knitro-5.0.0-student for the student edition) containing the following files and subdirectories:

INSTALL: A file containing installation instructions.

LICENSE_KNITRO: A file containing the KNITRO license agreement.

README: A file with instructions on how to get started using KNITRO.

KNITRO50-ReleaseNotes: A file containing 5.0 release notes.

get_machine_ID: An executable that identifies the machine ID, required for obtaining a Ziena license file.

doc: A directory containing KNITRO documentation, including this manual.

include: A directory containing the KNITRO header file knitro.h.

lib: A directory containing the KNITRO library files: libknitro.a and libknitro.so.

examples: A parent directory containing examples of how to use the KNITRO API in different programming languages (C, Fortran, Java).

knitroampl: A directory containing files, instructions, and an example model for using KNITRO with AMPL.

To activate KNITRO for your computer you will need a valid Ziena license file. For a single machine license, execute get_machine_ID, a program supplied with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Email the machine ID to `info@ziena.com` if purchased through Ziena. (If KNITRO was purchased through a distributor, then email the machine ID to your local distributor.) Ziena (or your local distributor) will then send a license file of the form `ziena_lic_name.txt`. The file name is arbitrary, so long as it begins with `"ziena_"`. The license file is made available to KNITRO in any one of the following ways:

- Put `ziena_lic_name.txt` in your `$HOME` directory.
- Put `ziena_lic_name.txt` in the directory in which you run your KNITRO application.
- Set the environment variable `ZIENA_LICENSE` to the location of the file (e.g., `export ZIENA_LICENSE=/home/subdirectory/ziena_lic_name.txt`).

If you have difficulty installing the license file, then set the environment variable `ZIENA_LICENSE_DEBUG` and the license manager will display helpful information. Type `export ZIENA_LICENSE_DEBUG=1` and then try running KNITRO again.

Before beginning, view the INSTALL, LICENSE_KNITRO and README files. To get started, build and test the programs provided inside the examples directory. Example problems are provided for the C, Fortran, and Java interfaces. We recommend understanding these examples and reading section 4 of this manual before proceeding with development of your own application interface. The knitroampl subdirectory contains instructions for using KNITRO with AMPL and an example of how to formulate a nonlinear optimization problem in AMPL.

The programs in examples/C usually build on Linux 32-bit platforms by simply typing `"gmake"` (on Linux 64-bit machines and Solaris the makefile must first be edited, as described in the makefile itself). However, you may see a long list of link errors similar to the following:

```
../lib/libknitro.a(.text+0x28808): In function ‘ktr_xeb4’:
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
../lib/libknitro.a(.text+0x28837): In function ‘ktr_xeb4’:
```

```
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
../lib/libknitro.a(.text+0x290b0): more undefined references to ‘std::__
default_alloc_template<true, 0>::deallocate(void*, unsigned int)’ follow
../lib/libknitro.a(.text+0x2a0ff): In function ‘ktr_x1150’:
: undefined reference to ‘std::basic_string<char, std::char_traits<char>
, std::allocator<char> >::_S_empty_rep_storage’
../lib/libknitro.a(.text+0x2a283): In function ‘ktr_x1150’:
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
```

This indicates an incompatibility between the libstdc++ library on your Linux distribution and the library that KNITRO was built with. The incompatibilities may be caused by name-mangling differences between versions of the gcc compiler, and by differences in the Application Binary Interface of the two Linux distributions. The best fix is for Ziena to rebuild the KNITRO binaries on the exact same Linux distribution of your target machine. If you see these errors, please contact Ziena `info@ziena.com` to correct the problem. These errors are seen only on Linux.

3 Using KNITRO with the AMPL modeling language

AMPL is a popular modeling language for optimization which allows users to represent their optimization problems in a user-friendly, readable, intuitive format. This makes ones job of formulating and modeling a problem much simpler. For a description of AMPL see [7] or visit the AMPL web site at:

```
http://www.ampl.com/
```

It is straightforward to use KNITRO with the AMPL modeling language. We assume in the following that the user has successfully installed AMPL and that the KNITRO/AMPL executable file `knitro-ampl` resides in the current directory or in a directory which is specified in ones `PATH` environment variable (such as a `bin` directory).

Inside of AMPL, to invoke the KNITRO solver type:

```
option solver knitro-ampl;
```

at the prompt. Likewise, to specify user options one would type, for example,

```
option knitro_options ''maxit=100 alg=2'';
```

The above command would set the maximum number of allowable iterations to 100 and choose the `Interior/CG` algorithm (see section 8). See Tables 1-2 for a summary of all available user specifiable options in KNITRO for use with AMPL. For more detail on these options see section 5. Note, that in section 5, user parameters for the callable library are of the form "`KTR_PARAM_NAME`". In AMPL, parameters are set using only the (lowercase) "`name`" as specified in Tables 1-2.

NOTE: AMPL will often perform a reordering of the variables and constraints defined in the AMPL model (and may also simplify the form of the problem using a presolve). The output printed by KNITRO will correspond to this reformulated problem. To view values of the variables and constraints in the order and form corresponding to the original AMPL model, use the AMPL `display` command.

Below is an example AMPL model and AMPL session which calls KNITRO to solve the problem:

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 && \text{(3.2a)}\\
\text{subject to} \quad & 8x_1 + 14x_2 + 7x_3 - 56 = 0 && \text{(3.2b)}\\
& x_1^2 + x_2^2 + x_3^2 - 25 \geq 0 && \text{(3.2c)}\\
& x_1, x_2, x_3 \geq 0 && \text{(3.2d)}
\end{aligned}$$

with initial point $x = [x_1, x_2, x_3] = [2, 2, 2]$.

Assume the AMPL model for the above problem is defined in a file called testproblem.mod which is shown below (this example is also included in the knitro-ampl directory that comes with the KNITRO distribution).

AMPL test program file testproblem.mod

```
#
# Example problem formulated as an AMPL model used
# to demonstate using KNITRO with AMPL.
#

# Define variables and enforce that they be non-negative.

var x{j in 1..3} >= 0;

# Objective function to be minimized.

minimize obj:

         1000 - x[1]^2 - 2*x[2]^2 - x[3]^2 - x[1]*x[2] - x[1]*x[3];

# Equality constraint.

s.t. c1: 8*x[1] + 14*x[2] + 7*x[3] - 56 = 0;

# Inequality constraint.

s.t. c2: x[1]^2 + x[2]^2 + x[3]^2 -25 >= 0;

data;

# Define initial point.

let x[1] := 2;
let x[2] := 2;
let x[3] := 2;
```

The above example displays the ease with which one can express an optimization problem in the AMPL modeling language. Below is the AMPL session used to solve this problem with KNITRO. In the example below we set `alg=2` (to use the Interior/CG algorithm), `opttol=1e-8` (to tighten the optimality stopping tolerance) and `outlev=3` (to print output at each major iteration). See section 7 for an explanation of the output.

AMPL Example

```
ampl: reset;
ampl: option solver knitro-ampl;
ampl: option knitro_options "alg=2 opttol=1e-8 outlev=3";
ampl: model testproblem.mod;
ampl: solve;
```

```
KNITRO 5.0: alg=2
opttol=1e-8
outlev=3

=====================================
             KNITRO 5.0.0
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
=====================================

algorithm:    2
opttol:       1e-08
outlev:       3

Problem Characteristics
-----------------------
Number of variables:                     3
    bounded below:                       3
    bounded above:                       0
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   2
    linear equalities:                   1
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              1
    range:                               0
Number of nonzeros in Jacobian:          6
Number of nonzeros in Hessian:           5

  Iter      Objective      Feas err    Opt err     ||Step||    CG its
--------  -------------  ----------  ----------  ----------  -------
       0    9.760000e+02   1.300e+01
       1    9.688061e+02   7.190e+00   6.228e+00   1.513e+00        1
       2    9.397651e+02   1.946e+00   2.988e+00   5.659e+00        1
       3    9.323614e+02   1.659e+00   5.003e-03   1.238e+00        2
       4    9.361994e+02   1.421e-14   9.537e-03   2.395e-01        2
       5    9.360402e+02   7.105e-15   1.960e-03   1.591e-02        2
       6    9.360004e+02   0.000e+00   1.597e-05   4.017e-03        1
       7    9.360000e+02   7.105e-15   1.945e-07   3.790e-05        2
       8    9.360000e+02   0.000e+00   1.925e-09   3.990e-07        1

EXIT: LOCALLY OPTIMAL SOLUTION FOUND.
```

```
Final Statistics
----------------
Final objective value               =   9.36000000040000e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   1.92e-09 / 1.20e-10
# of iterations (major / minor)     =          8 /        8
# of function evaluations           =          9
# of gradient evaluations           =          9
# of Hessian evaluations            =          8
Total program time (secs)           =       0.00321 (     0.000 CPU time)

================================================================================

KNITRO 5.0: LOCALLY OPTIMAL SOLUTION FOUND.
objective 9.360000e+02; feasibility error 0.000000e+00
8 major iterations; 9 function evaluations
ampl:
```

For descriptions of the KNITRO output see section 7. To display the final solution variables `x` and the objective value `obj` through AMPL, use the AMPL `display` command as follows.

```
ampl: display x;
x [*] :=
1  1.92477e-09
2  6.31331e-10
3  8
;

ampl: display obj;
obj = 936
```

Solving Mathematical Programs with Equilibrium Constraints (MPECs)

KNITRO is able to effectively solve problems with equilibrium (or complementarity) constraints (MPECs) through the AMPL interface. A complementarity constraint is a constraint which enforces that two variables are *complementary* to each other, i.e., the variables x_1 and x_2 are complementary if the following conditions hold

$$x_1 \times x_2 = 0, \quad x_1 \geq 0, \quad x_2 \geq 0. \tag{3.3}$$

The condition above, is sometimes expressed more compactly as

$$0 \leq x_1 \quad \perp \quad x_2 \geq 0.$$

These constraints must be formulated in a particular way through AMPL in order for KNITRO to effectively deal with them. In particular, complementarity constraints should be modeled using the AMPL `complements` command, e.g.,

```
0 <= x1 complements x2 >= 0;
```

and they *must* be formulated as one variable complementary to another variable. They *may not* be formulated as a function complementarity to a variable or a function complementary to a function. If the complementarity involves a function $F(x)$, for example,

$$0 \leq F(x) \quad \perp \quad x \geq 0,$$

the user should reformulate the AMPL model by adding a slack variable, as shown below, so that it is formulated as a variable complementary to another variable.

```
var x; var s;
...
constraint_name_a: F(x) = s;
constraint_name_b: 0 <= s complements x >= 0;
```

See section 8 for KNITRO algorithm descriptions and section 9 for other KNITRO special features. A user running AMPL can skip section 4.

OPTION	DESCRIPTION	DEFAULT
alg	optimization algorithm used: 0: automatic algorithm selection 1: Interior/Direct algorithm 2: Interior/CG algorithm 3: Active algorithm	0
barrule	barrier paramater update rule: 0: automatic barrier rule chosen 1: monotone decrease rule 2: adaptive rule based on centrality measure 3: probing rule 4: safeguarded Mehrotra predictor-corrector type rule 5: Mehrotra predictor-corrector type rule 6: rule based on minimizing a quality function	0
debug	enable debugging output: 0: no extra debugging 1: help debug solution of the problem 2: help debug execution of the solver	0
delta	initial trust region radius scaling	1.0e0
feasible	0: allow for infeasible iterates 1: feasible version of KNITRO	0
feasmodetol	tolerance for entering feasible mode	1.0e-4
feastol	feasibility termination tolerance (relative)	1.0e-6
feastolabs	feasibility termination tolerance (absolute)	0.0e-0
gradopt	gradient computation method: 1: use exact gradients (only option available for AMPL interface)	1
hessopt	Hessian (Hessian-vector) computation method: 1: use exact Hessian 2: use dense quasi-Newton BFGS Hessian approximation 3: use dense quasi-Newton SR1 Hessian approximation 4: compute Hessian-vector products via finite differencing 5: compute exact Hessian-vector products 6: use limited-memory BFGS Hessian approximation	1
honorbnds	0: allow bounds to be violated during the optimization 1: enforce satisfaction of simple bounds always	0
initpt	0: do not use any initial point strategies 1: use initial point strategy	0
lmsize	number of limited-memory pairs stored in LBFGS approach	10

Table 1: KNITRO user specifiable options for AMPL.

OPTION	DESCRIPTION	DEFAULT
lpsolver	1: use internal LP solver in active-set algorithm 2: use ILOG-CPLEX LP solver in active-set algorithm (requires valid CPLEX license)	1
maxcgit	maximum allowable conjugate gradient (CG) iterations: 0: automatically set based on the problem size n: maximum of n CG iterations per minor iteration	0
maxcrossit	maximum number of allowable crossover iterations	0
maxit	maximum number of iterations before terminating	10000
maxtime_cpu	maximum CPU time in seconds before terminating	1.0e8
maxtime_real	maximum real time in seconds before terminating	1.0e8
ms_maxsolves	maximum KNITRO solves for multistart	1
mu	initial barrier parameter value	1.0e-1
multistart	enable the multistart feature	0
objrange	allowable objective function range	1.0e20
opttol	optimality termination tolerance (relative)	1.0e-6
opttolabs	optimality termination tolerance (absolute)	0.0e-0
outlev	printing output level: 0: no printing 1: just print summary information 2: print information every 10 major iterations 3: print information at each major iteration 4: print information at each major and minor iteration 5: also print final (primal) variables 6: also print final constraint values and Lagrange multipliers	2
outmode	where to direct output: 0: print to standard out (e.g., screen) 1: print to file 'knitro.log' 2: both screen and file 'knitro.log'	0
pivot	initial pivot threshold for matrix factorizations	1.0e-8
scale	0: do not scale the problem 1: perform automatic scaling of functions	1
shiftinit	0: do not shift the initial point 1: shift the initial point to satisfy the bounds	1
soc	0: do not allow second order correction steps 1: selectively try second order correction steps 2: always try second order correction steps	1
xtol	stepsize termination tolerance	1.0e-15

Table 2: KNITRO user specifiable options for AMPL (continued).

4 The KNITRO callable library

This section includes information on how to embed and call the KNITRO optimizer from inside a program. KNITRO is written in C and C++, with a well-documented application programming interface (API) defined in the file knitro.h. The KNITRO 5.0 product contains example interfaces written in various programming languages under the directory /examples. These are briefly discussed in the following sections. Each example consists of a main driver program coded in the given language that defines an optimization problem and invokes KNITRO to solve it. Examples also contain a makefile illustrating how to link the KNITRO library with the target language driver program.

In all languages KNITRO runs as a thread-safe module, which means that the calling program can create multiple instances of a KNITRO solver in different threads, each instance solving a different problem. This is useful in a multiprocessing environment; for instance, in a web application server.

4.1 KNITRO in a C application

The KNITRO callable library can be used to solve an optimization problem coded in the C language through a sequence of four basic function calls:

- `KTR_new()`: create a new KNITRO solver context pointer, allocating resources
- `KTR_init_problem()`: load the problem definition into the KNITRO solver
- `KTR_solve()`: solve the problem
- `KTR_free()`: delete the KNITRO context pointer, releasing allocated resources

The complete C language API is defined in the file knitro.h, provided in the installation under the /include directory. Functions for setting and getting user options are described in sections 5.2 and 5.3. Functions for retrieving KNITRO results are described in section 7.2 and illustrated in the examples/C files. The remainder of this section describes in detail the four basic function calls.

```
KTR_context_ptr KTR_new (void)
```

This function must be called first. It returns a pointer to an object (the KNITRO "context pointer") that is used in all other calls. If you enable KNITRO with the Ziena floating network license handler, then this call also checks out a license and reserves it until `KTR_free()` is called with the context pointer, or the program ends. The contents of the context pointer should never be modified by a calling program.

```
int KTR_free (KTR_context_ptr * kc_handle)
```

This function should be called last and will free the context pointer. The address of the context pointer is passed so that KNITRO can set it to NULL after freeing all memory. This prevents the application from mistakenly calling KNITRO functions after the context pointer has been freed.

The C interface for KNITRO requires the application to define an optimization problem (1.1) in the following general format:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && \text{(4.4a)} \\ \text{subject to} \quad & \texttt{cLoBnds} \le c(x) \le \texttt{cUpBnds} && \text{(4.4b)} \\ & \texttt{xLoBnds} \le x \le \texttt{xUpBnds} && \text{(4.4c)} \end{aligned}$$

where `cLoBnds` and `cUpBnds` are vectors of length m, and `xLoBnds` and `xUpBnds` are vectors of length n. If constraint i is an equality constraint, set $\texttt{cLoBnds}[i] = \texttt{cUpBnds}[i]$. If constraint i is unbounded from below or above, set $\texttt{cLoBnds}[i]$ or $\texttt{cUpBnds}[i]$ to the value `-KTR_INFBOUND` or `KTR_INFBOUND`, respectively. This constant is defined in knitro.h and stands for ∞ in the KNITRO code.

To use KNITRO the application must provide routines for evaluating the objective $f(x)$ and constraint functions $c(x)$, the first derivatives (gradients of $f(x)$ and $c(x)$), and optionally, the second derivatives (Hessian of the Lagrangian). First derivatives in the C language API are denoted by `objGrad` and `jac`, where $\texttt{objGrad} = \nabla f(x)$, and `jac` is the $m \times n$ Jacobian matrix of constraint gradients such that the i-th row equals $\nabla c_i(x)$.

The ability to provide exact first derivatives is essential for efficient and reliable performance. Packages like ADOL-C and ADIFOR can help in generating code with derivatives. If the user is unable or unwilling to provide exact first derivatives, KNITRO provides an option that computes approximate first derivatives using finite-differencing (see section 9.1).

Exact second derivatives are less important, as KNITRO provides several options that substitute quasi-Newton approximations for the Hessian (see section 9.2). However, the ability to provide exact second derivatives often dramatically improves the performance of KNITRO.

```
int KTR_init_problem(  KTR_context_ptr kc,
                       int n,
                       int objGoal,
                       int objType,
                       double * xLoBnds,
                       double * xUpBnds,
                       int m,
                       int * cType,
                       double * cLoBnds,
                       double * cUpBnds,
                       int nnzJ,
                       int * jacIndexVars,
                       int * jacIndexCons,
                       int nnzH,
                       int * hessIndexRows,
                       int * hessIndexCols,
                       double * xInitial,
                       double * lambdaInitial )
```

This function passes the optimization problem definition to KNITRO, where it is copied and stored internally until `KTR_free()` is called. Once initialized, the problem may be solved any number of times with different user options or initial points (see the `KTR_restart()` call below). Array arguments passed

to KTR_init_problem() are not referenced again and may be freed or reused if desired. In the description below, some programming macros are mentioned as alternatives to fixed numeric constants; e.g., KTR_OBJGOAL_MINIMIZE. These macros are defined in knitro.h

Arguments:

KTR_context_ptr kc: is the KNITRO context pointer.

int n: is a scalar specifying the number of variables in the problem; i.e., the length of x in (4.4).

int objGoal: is the optimization goal.

- 0: if the goal is to minimize the objective function (KTR_OBJGOAL_MINIMIZE)
- 1: if the goal is to maximize the objective function (KTR_OBJGOAL_MAXIMIZE)

int objType: is a scalar that describes the type of objective function $f(x)$ in (4.4).

- 0: if $f(x)$ is a nonlinear function or its type is unknown (KTR_OBJTYPE_GENERAL)
- 1: if $f(x)$ is a linear function (KTR_OBJTYPE_LINEAR)
- 2: if $f(x)$ is a quadratic function (KTR_OBJTYPE_QUADRATIC)

double * xLoBnds: is an array of length n specifying the lower bounds on x. xLoBnds[i] must be set to the lower bound of the corresponding i-th variable x_i. If the variable has no lower bound, set xLoBnds[i] to be -KTR_INFBOUND (defined in knitro.h).

double * xUpBnds: is an array of length n specifying the upper bounds on x. xUpBnds[i] must be set to the upper bound of the corresponding i-th variable x_i. If the variable has no upper bound, set xUpBnds[i] to be KTR_INFBOUND (defined in knitro.h).

int m: is a scalar specifying the number of constraints $c(x)$ in (4.4).

int * cType: is an array of length m that describes the types of the constraint functions $c(x)$ in (4.4).

- 0: if cType[i] is a nonlinear function or its type is unknown (KTR_CONTYPE_GENERAL)
- 1: if cType[i] is a linear function (KTR_CONTYPE_LINEAR)
- 2: if cType[i] is a quadratic function (KTR_CONTYPE_QUADRATIC)

double * cLoBnds: is an array of length m specifying the lower bounds on the constraints $c(x)$ in (4.4). cLoBnds[i] must be set to the lower bound of the corresponding i-th constraint. If the constraint has no lower bound, set cLoBnds[i] to be -KTR_INFBOUND (defined in knitro.h). If the constraint is an equality, then cLoBnds[i] should equal cUpBnds[i].

double * cUpBnds: is an array of length m specifying the upper bounds on the constraints $c(x)$ in (4.4). cUpBnds[i] must be set to the upper bound of the corresponding i-th constraint. If the constraint has no upper bound, set cUpBnds[i] to be KTR_INFBOUND (defined in knitro.h). If the constraint is an equality, then cLoBnds[i] should equal cUpBnds[i].

int nnzJ: is a scalar specifying the number of nonzero elements in the sparse constraint Jacobian. See section 4.6 for an example.

`int * jacIndexVars:` is an array of length `nnzJ` specifying the variable indices of the constraint Jacobian nonzeroes. If `jacIndexVars`[i]=j, then `jac`[i] refers to the j-th variable, where `jac` is the array of constraint Jacobian nonzero elements passed in the call `KTR_solve()`.

`jacIndexCons`[i] and `jacIndexVars`[i] determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element `jac`[i]. See section 4.6 for an example.

NOTE: C array numbering starts with index 0. Therefore, the j-th variable x_j maps to array element `x[j]`, and $0 \leq$ `j` $<$ `n`.

`int * jacIndexCons:` is an array of length `nnzJ` specifying the constraint indices of the constraint Jacobian nonzeroes. If `jacIndexCons`[i]=k, then `jac`[i] refers to the k-th constraint, where `jac` is the array of constraint Jacobian nonzero elements passed in the call `KTR_solve()`.

`jacIndexCons`[i] and `jacIndexVars`[i] determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element `jac`[i]. See section 4.6 for an example.

NOTE: C array numbering starts with index 0. Therefore, the k-th constraint c_k maps to array element `c[k]`, and $0 \leq$ `k` $<$ `m`.

`int nnzH:` is a scalar specifying the number of nonzero elements in the sparse Hessian of the Lagrangian. Only nonzeroes in the upper triangle (including diagonal nonzeroes) should be counted. See section 4.6 for an example.

NOTE: If user option "hessopt" is not set to `KTR_HESSOPT_EXACT`, then Hessian nonzeroes will not be used (see section 5.1). In this case, set `nnzH=0`, and pass NULL pointers for `hessIndexRows` and `hessIndexCols`.

`int * hessIndexRows:` is an array of length `nnzH` specifying the row number indices of the Hessian nonzeroes.

`hessIndexRows`[i] and `hessIndexCols`[i] determine the row numbers and the column numbers, respectively, of the nonzero Hessian element `hess`[i], where `hess` is the array of Hessian elements passed in the call `KTR_solve()`. See section 4.6 for an example.

NOTE: Row numbers are in the range 0 .. `n` $-$ 1.

`int * hessIndexCols:` is an array of length `nnzH` specifying the column number indices of the Hessian nonzeroes.

`hessIndexRows`[i] and `hessIndexCols`[i] determine the row numbers and the column numbers, respectively, of the nonzero Hessian element `hess`[i], where `hess` is the array of Hessian elements passed in the call `KTR_solve()`. See section 4.6 for an example.

NOTE: Column numbers are in the range 0 .. `n` $-$ 1.

`double * xInitial:` is an array of length `n` containing an initial guess of the solution vector x. If the application prefers to let KNITRO make an initial guess, then pass a NULL pointer for `xInitial`.

`double * lambdaInitial:` is an array of length `m + n` containing an initial guess of the Lagrange multipliers for the constraints $c(x)$ (4.4*b*) and bounds on the variables x (4.4*c*). The first `m` components of `lambdaInitial` are multipliers corresponding to the constraints specified in $c(x)$, while the last `n` components are multipliers corresponding to the bounds on x. If the application prefers to let KNITRO make an initial guess, then pass a NULL pointer for `lambdaInitial`.

To solve the nonlinear optimization problem (4.4), KNITRO needs the application to supply information at various trial points. KNITRO specifies a trial point with a new vector of variable values x, and sometimes a corresponding vector of Lagrange multipliers λ. At a trial point, KNITRO may ask the application to:

`KTR_RC_EVALFC`:	Evaluate f and c at x.
`KTR_RC_EVALGA`:	Evaluate ∇f and ∇c at x.
`KTR_RC_EVALH`:	Evaluate the Hessian matrix of the problem at x and λ.
`KTR_RC_EVALHV`:	Evaluate the Hessian matrix times a vector v at x and λ.

The constants `KTR_RC_*` are return codes defined in knitro.h.

The KNITRO C language API has two modes of operation for obtaining problem information: "**callback**" and "**reverse communication**". With callback mode the application provides C language function pointers that KNITRO may call to evaluate the functions, gradients, and Hessians. With reverse communication, the function `KTR_solve()` returns one of the constants listed above to tell the application what it needs, and then waits to be called again with the new problem information. For more details, see section 9.8 (callback mode) and section 9.7 (reverse communication mode). Both modes use `KTR_solve()`.

```
int KTR_solve(  KTR_context_ptr kc,    /*input*/
                double * x,            /*output*/
                double * lambda,       /*output*/
                int evalStatus,        /*input, reverse comm only*/
                double * obj,          /*input and output*/
                double * c,            /*input, reverse comm only*/
                double * objGrad,      /*input, reverse comm only*/
                double * jac,          /*input, reverse comm only*/
                double * hess,         /*input, reverse comm only*/
                double * hessVector,   /*input, output, rev comm*/
                void * userParams )    /*input, callback only*/
```

Arguments:

`KTR_context_ptr kc:` is the KNITRO context pointer.

`double * x:` is an array of length `n` output by KNITRO. If `KTR_solve()` returns zero, then `x` contains the solution.

Reverse communications mode: `x` contains the value of unknowns at which KNITRO needs more problem information. If user option "newpoint" is enabled (see section 5.1) and `KTR_solve()` returns `KTR_RC_NEWPOINT`, then `x` contains a newly accepted iterate, but not the final solution.

`double * lambda:` is an array of length `m + n` output by KNITRO. If `KTR_solve()` returns zero, then `lambda` contains the multiplier values at the solution. The first `m` components of `lambda` are multipliers corresponding to the constraints specified in $c(x)$, while the last `n` components are multipliers corresponding to the bounds on x.

Reverse communications mode: `lambda` contains the value of multipliers at which KNITRO needs more problem information.

`int evalStatus:` is a scalar input to KNITRO used only in **reverse communications mode**. A value of zero means the application successfully computed the problem information requested by KNITRO at `x` and `lambda`. A nonzero value means the application failed to compute problem information (e.g., if a function is undefined at the requested value `x`).

`double * obj:` is a scalar holding the value of $f(x)$ at the current `x`. If `KTR_solve()` returns zero, then `obj` contains the value of the objective function $f(x)$ at the solution.

Reverse communications mode: if `KTR_solve()` returns `KTR_RC_EVALFC`, then `obj` must be filled with the value of $f(x)$ computed at `x` before `KTR_solve()` is called again.

`double * c:` is an array of length `m` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALFC`, then `c` must be filled with the value of $c(x)$ computed at `x` before `KTR_solve()` is called again.

`double * objGrad:` is an array of length `n` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALGA`, then `objGrad` must be filled with the value of $\nabla f(x)$ computed at `x` before `KTR_solve()` is called again.

`double * jac:` is an array of length `nnzJ` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALGA`, then `jac` must be filled with the constraint Jacobian $\nabla c(x)$ computed at `x` before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`.

`double * hess:` is an array of length `nnzH` used only in **reverse communications mode**, and only if option "hessopt" is set to `KTR_HESSOPT_EXACT` (see section 5.1). If `KTR_solve()` returns `KTR_RC_EVALH`, then `hess` must be filled with the Hessian of the Lagrangian computed at `x` and `lambda` before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`.

`double * hessVector:` is an array of length `n` used only in **reverse communications mode**, and only if option "hessopt" is set to `KTR_HESSOPT_PRODUCT` (see section 5.1). If `KTR_solve()` returns `KTR_RC_EVALHV`, then the Hessian of the Lagrangian at `x` and `lambda` should be multiplied by `hessVector`, and the result placed in `hessVector` before `KTR_solve()` is called again.

`void * userParams:` is a pointer to a structure used only in **callback mode**. The pointer is provided so the application can pass additional parameters needed for its callback routines. If the application needs no additional parameters, then pass a NULL pointer. See section 9.8 for more details.

Return Value:

The return value of `KTR_solve()` specifies the final exit code from the optimization process. If the return value is 0 or negative, then KNITRO has finished solving. In **reverse communications mode** the return value may be positive, in which case it specifies a request for additional problem information, after which the application should call KNITRO again. A detailed description of all the possible return values is given in the appendix.

```
int KTR_restart(  KTR_context_ptr kc,
                  double * x,
                  double * lambda )
```

This function can be called to modify user options or initial values before starting another `KTR_solve()` sequence. `KTR_restart()` prepares KNITRO for a new `KTR_solve()` sequence, but is not allowed to change the problem definition established in `KTR_init_problem()`. A sample program in examples/C uses `KTR_restart()` to solve the same problem from the same start point, but varying the interior point "barrule" option over all its possible values.

4.2 Examples of calling in C

The KNITRO distribution comes with several C language programs in the directory examples/C. The instructions in examples/C/README.txt explain how to compile and run the examples. This section overviews the coding of driver programs, but the working examples provide more complete detail.

Consider the following nonlinear optimization problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100-(x_2-x_1^2)^2+(1-x_1)^2 && \text{(4.5a)}\\ \text{subject to} \quad & 1 \le x_1 x_2 && \text{(4.5b)}\\ & 0 \le x_1+x_2^2 && \text{(4.5c)}\\ & x_1 \le 0.5. && \text{(4.5d)} \end{aligned}$$

This problem is coded as examples/C/problemHS15.c [9].

Every driver starts by allocating a new KNITRO solver instance and checking that it succeeded (the only reason `KTR_new()` might fail is if the Ziena license check fails):

```
#include "knitro.h"

/*... Include other headers, define main() ...*/

KTR_context  *kc;

/*... Declare other local variables ...*/

/*---- CREATE A NEW KNITRO SOLVER INSTANCE. */
kc = KTR_new();
if (kc == NULL)
    {
    printf ("Failed to find a Ziena license.\n");
```

```
    return( -1 );
    }
```

The next task is to load the optimization problem definition into the solver using `KTR_init_problem()`. The problem has 2 unknowns and 2 constraints, and it is easily seen that all first and second partial derivatives are generally nonzero. The code segment below captures the problem definition and passes it to KNITRO:

```
/*---- DEFINE PROBLEM SIZES. */
n = 2;
m = 2;
nnzJ = 4;
nnzH = 3;

/*... allocate memory for xLoBnds, xUpBnds, etc. ...*/

/*---- DEFINE THE OBJECTIVE FUNCTION AND VARIABLE BOUNDS. */
objType = KTR_OBJTYPE_GENERAL;
objGoal = KTR_OBJGOAL_MINIMIZE;
xLoBnds[0] = -KTR_INFBOUND;
xLoBnds[1] = -KTR_INFBOUND;
xUpBnds[0] = 0.5;
xUpBnds[1] = KTR_INFBOUND;

/*---- DEFINE THE CONSTRAINT FUNCTIONS. */
cType[0] = KTR_CONTYPE_QUADRATIC;
cType[1] = KTR_CONTYPE_QUADRATIC;
cLoBnds[0] = 1.0;
cLoBnds[1] = 0.0;
cUpBnds[0] = KTR_INFBOUND;
cUpBnds[1] = KTR_INFBOUND;

/*---- PROVIDE FIRST DERIVATIVE STRUCTURAL INFORMATION. */
jacIndexCons[0] = 0;
jacIndexCons[1] = 0;
jacIndexCons[2] = 1;
jacIndexCons[3] = 1;
jacIndexVars[0] = 0;
jacIndexVars[1] = 1;
jacIndexVars[2] = 0;
jacIndexVars[3] = 1;

/*---- PROVIDE SECOND DERIVATIVE STRUCTURAL INFORMATION. */
hessIndexRows[0] = 0;
hessIndexRows[1] = 0;
hessIndexRows[2] = 1;
hessIndexCols[0] = 0;
hessIndexCols[1] = 1;
```

```
    hessIndexCols[2] = 1;

    /*---- CHOOSE AN INITIAL START POINT. */
    xInitial[0] = -2.0;
    xInitial[1] =  1.0;

    /*---- INITIALIZE KNITRO WITH THE PROBLEM DEFINITION. */
    nStatus = KTR_init_problem (kc, n, objGoal, objType,
                                xLoBnds, xUpBnds,
                                m, cType, cLoBnds, cUpBnds,
                                nnzJ, jacIndexVars, jacIndexCons,
                                nnzH, hessIndexRows, hessIndexCols,
                                xInitial, NULL);
    if (nStatus != 0)
        { /*... an error occurred ...*/ }

    /*... free xLoBnds, xUpBnds, etc. ...*/
```

Assume for simplicity that the user writes three routines for computing problem information. In examples/C/problemHS15.c these are named `computeFC`, `computeGA`, and `computeH`. To write a driver program using **callback** mode, simply wrap each evaluation routine in a function that matches the `KTR_callback()` prototype defined in knitro.h. Note that all three wrappers use the same prototype. This is in case the application finds it convenient to combine some of the evaluation steps, as demonstrated in one of the example programs.

```
/*-----------------------------------------------------------------*/
/*     FUNCTION callbackEvalFC                                     */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *  Only "obj" and "c" are modified.
 */
int  callbackEvalFC (const int             evalRequestCode,
                     const int             n,
                     const int             m,
                     const int             nnzJ,
                     const int             nnzH,
                     const double * const  x,
                     const double * const  lambda,
                           double * const  obj,
                           double * const  c,
                           double * const  objGrad,
                           double * const  jac,
                           double * const  hessian,
                           double * const  hessVector,
                           void   *        userParams)
{
    if (evalRequestCode != KTR_RC_EVALFC)
```

```
      {
      printf ("*** callbackEvalFC incorrectly called with eval code %d\n",
              evalRequestCode);
      return( -1 );
      }

    /*---- IN THIS EXAMPLE, CALL THE ROUTINE IN problemDef.h. */
    *obj = computeFC (x, c);
    return( 0 );
}


/*-----------------------------------------------------------------*/
/*     FUNCTION callbackEvalGA                                     */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *  Only "objGrad" and "jac" are modified.
 */

    /*... similar implementation to callbackEvalFC ...*/



/*-----------------------------------------------------------------*/
/*     FUNCTION callbackEvalH                                      */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *  Only "hessian" is modified.
 */

    /*... similar implementation to callbackEvalFC ...*/
```

Back in the main program each wrapper function is registered as a callback to KNITRO, and then `KTR_solve()` is invoked to find the solution:

```
    /*---- REGISTER THE CALLBACK FUNCTIONS THAT PERFORM PROBLEM EVALS.
     *---- THE HESSIAN CALLBACK ONLY NEEDS TO BE REGISTERED FOR SPECIFIC
     *---- HESSIAN OPTIONS (E.G., IT IS NOT REGISTERED IF THE OPTION FOR
     *---- BFGS HESSIAN APPROXIMATIONS IS SELECTED).
     */
    if (KTR_set_func_callback (kc, &callbackEvalFC) != 0)
        exit( -1 );
    if (KTR_set_grad_callback (kc, &callbackEvalGA) != 0)
        exit( -1 );
    if ((nHessOpt == KTR_HESSOPT_EXACT) ||
        (nHessOpt == KTR_HESSOPT_PRODUCT))
        {
```

```
    if (KTR_set_hess_callback (kc, &callbackEvalHess) != 0)
        exit( -1 );
    }

/*---- SOLVE THE PROBLEM.
 */
nStatus = KTR_solve (kc, x, lambda, 0, &obj,
                     NULL, NULL, NULL, NULL, NULL, NULL);
if (nStatus != 0)
  printf ("KNITRO failed to solve the problem, final status = %d\n",
          nStatus);

/*---- DELETE THE KNITRO SOLVER INSTANCE. */
KTR_free (&kc);
```

To write a driver program using **reverse communications** mode, set up a loop that calls `KTR_solve()` and then computes the requested problem information. The loop continues until `KTR_solve()` returns zero (success), or a negative error code:

```
/*---- SOLVE THE PROBLEM.  IN REVERSE COMMUNICATIONS MODE, KNITRO
 *---- RETURNS WHENEVER IT NEEDS MORE PROBLEM INFO.  THE CALLING
 *---- PROGRAM MUST INTERPRET KNITRO'S RETURN STATUS AND CONTINUE
 *---- SUPPLYING PROBLEM INFORMATION UNTIL KNITRO IS COMPLETE.
 */
while (1)
    {
    nStatus = KTR_solve (kc, x, lambda, evalStatus, &obj, c,
                         objGrad, jac, hess, hvector, NULL);

    if      (nStatus == KTR_RC_EVALFC)
        /*---- KNITRO WANTS obj AND c EVALUATED AT THE POINT x. */
        obj = computeFC (x, c);
    else if (nStatus == KTR_RC_EVALGA)
        /*---- KNITRO WANTS objGrad AND jac EVALUATED AT x. */
        computeGA (x, objGrad, jac);
    else if (nStatus == KTR_RC_EVALH)
        /*---- KNITRO WANTS hess EVALUATED AT (x, lambda). */
        computeH (x, lambda, hess);
    else
        /*---- IN THIS DRIVE, OTHER STATUS CODES MEAN KNITRO IS
               FINISHED. */
        break;

    /*---- ASSUME THAT PROBLEM EVALUATION IS ALWAYS SUCCESSFUL.
     *---- IF A FUNCTION OR ITS DERIVATIVE COULD NOT BE EVALUATED
     *---- AT THE GIVEN (x, lambda), THEN SET evalStatus = 1 BEFORE
```

```
         *---- CALLING KTR_solve AGAIN. */
        evalStatus = 0;
        }

    if (nStatus != 0)
        printf ("KNITRO failed to solve the problem, final status = %d\n",
                nStatus);

    /*---- DELETE THE KNITRO SOLVER INSTANCE. */
    KTR_free (&kc);
```

This completes the brief overview of creating driver programs to run KNITRO in C. Again, more details and options are demonstrated in the programs located in examples/C. Outputs produced by KNITRO are discussed in section 7.

4.3 KNITRO in a C++ application

Calling KNITRO from a C++ application requires little or no modification of the C examples in the previous section. The KNITRO header file knitro.h already includes `extern C` statements to export KNITRO functions to C++.

4.4 KNITRO in a Java application

Calling KNITRO from a Java application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Java version of `KTR_init_problem()` is called. Java `int` and `double` types map directly to their C counterparts. Having defined the optimization problem, the Java version of `KTR_solve()` is called in **reverse communications mode**.

The KNITRO distribution provides a Java Native Interface (JNI) wrapper for the KNITRO callable library functions defined in knitro.h. The file examples/Java/KnitroJava.java is a Java class that provides access to the JNI functions. It loads lib/JNI-knitro.so, a JNI-enabled form of the KNITRO binary. In this way Java applications can create a KNITRO solver instance and call methods on the object. The call sequence is almost exactly the same as C applications that call knitro.h functions with a `KTR_context` reference. The JNI form of KNITRO is thread-safe, which means that a Java application can create multiple instances of a KNITRO solver in different threads, each instance solving a different problem. This feature might be important in an application that is deployed on a web server.

An example Java program is provided in examples/Java. The code will not be reproduced here, as it closely mirrors the structural form of the C reverse communications example described in section 4.2.

4.5 KNITRO in a Fortran application

Calling KNITRO from a Fortran application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Fortran version of `KTR_init_problem()` is called. Fortran `integer` and `double precision` types map directly to C `int` and `double` types. Having defined the optimization problem, the Fortran version of `KTR_solve()` is called in **reverse communications mode**.

Fortran applications require wrapper functions written in C to (1) isolate the `KTR_context` structure, which has no analog in unstructured Fortran, (2) convert C function names into names recognized by the Fortran linker, and (3) renumber array indices to start from zero (the C convention used by KNITRO) if the application follows the Fortran convention of starting from one. The wrapper functions can be called from Fortran with exactly the same arguments as their C language counterparts, except for the omission of the `KTR_context` argument.

An example Fortran program and set of C wrappers is provided in examples/Fortran. The code will not be reproduced here, as it closely mirrors the structural form of the C reverse communications example described in section 4.2. The example loads the matrix sparsity of the optimization problem with indices that start numbering from zero, and therefore requires no conversion from the Fortran convention of numbering from one. The C wrappers provided are sufficient for the simple example, but do not implement all the functionality of the KNITRO callable library. Users are free to write their own C wrapper routines, or extend the example wrappers as needed.

Ziena developers are available to port customer applications to any platform or compiler. The KNITRO 5.0 examples/Fortran files were compiled and tested with:

- `g77` on Linux
- Intel Visual Fortran 9 on Windows
- `f95` on Solaris (SPARC)

4.6 Specifying the Jacobian and Hessian matrices in sparse form

An important issue in using the KNITRO callable library is the ability of the user to specify the Jacobian matrix of the constraints and the Hessian of the Lagrangian function (when using exact Hessians) in sparse form. Below we give an example of how to do this.

Example

Assume we want to use KNITRO to solve the following problem

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & x_0 + x_1 x_2^3 && \text{(4.6a)}\\
\text{subject to} \quad & \cos(x_0) = 0.5 && \text{(4.6b)}\\
& 3 \le x_0^2 + x_1^2 \le 8 && \text{(4.6c)}\\
& x_0 + x_1 + x_2 \le 10 && \text{(4.6d)}\\
& x_0, x_1, x_2 \ge 1. && \text{(4.6e)}
\end{aligned}$$

Referring to (4.4), we have

$$\begin{aligned}
f(x) &= x_0 + x_1 x_2^3 && \text{(4.7)}\\
c_0(x) &= \cos(x_0) && \text{(4.8)}\\
c_1(x) &= x_0^2 + x_1^2 && \text{(4.9)}\\
c_2(x) &= x_0 + x_1 + x_2. && \text{(4.10)}\\
& && \text{(4.11)}
\end{aligned}$$

Computing the Sparse Jacobian Matrix

The gradients (first derivatives) of the objective and constraint functions are given by

$$\nabla f(x) = \begin{bmatrix} 1 \\ x_2^3 \\ 3x_1x_2^2 \end{bmatrix}, \nabla c_0(x) = \begin{bmatrix} -\sin(x_0) \\ 0 \\ 0 \end{bmatrix}, \nabla c_1(x) = \begin{bmatrix} 2x_0 \\ 2x_1 \\ 0 \end{bmatrix}, \nabla c_2(x) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

The constraint Jacobian matrix $J(x)$ is the matrix whose rows store the (transposed) constraint gradients, i.e.,

$$J(x) = \begin{bmatrix} \nabla c_0(x)^T \\ \nabla c_1(x)^T \\ \nabla c_2(x)^T \end{bmatrix} = \begin{bmatrix} -\sin(x_0) & 0 & 0 \\ 2x_0 & 2x_1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

In KNITRO, the array `objGrad` stores all of the elements of $\nabla f(x)$, while the arrays `jac`, `jacIndexCons`, and `jacIndexVars` store information concerning *only the nonzero* elements of $J(x)$. The array `jac` stores the nonzero values in $J(x)$ evaluated at the current solution estimate x, `jacIndexCons` stores the constraint function (or row) indices corresponding to these values and `jacIndexVars` stores the variable (or column) indices corresponding to these values. There is no restriction on the order in which these elements are stored, however, it is common to store the nonzero elements of $J(x)$ in column-wise fashion. For the example above, the number of nonzero elements `nnzJ` in $J(x)$ is 6, and these arrays would be specified as follows in column-wise order.

```
jac[0] = -sin(x[0]);  jacIndexCons[0] = 0; jacIndexVars[0] = 0;
jac[1] = 2*x[0];      jacIndexCons[1] = 1; jacIndexVars[1] = 0;
jac[2] = 1;           jacIndexCons[2] = 2; jacIndexVars[2] = 0;
jac[3] = 2*x[1];      jacIndexCons[3] = 1; jacIndexVars[3] = 1;
jac[4] = 1;           jacIndexCons[4] = 2; jacIndexVars[4] = 1;
jac[5] = 1;           jacIndexCons[5] = 2; jacIndexVars[5] = 2;
```

The values of `jac` depend on the value of x and may change while the values of `jacIndexCons` and `jacIndexVars` are constant.

Computing the Sparse Hessian Matrix

The Hessian of the Lagrangian matrix is defined as

$$H(x,\lambda) = \nabla^2 f(x) + \sum_{i=0..m-1} \lambda_i \nabla^2 c(x)_i, \tag{4.12}$$

where λ is the vector of Lagrange multipliers (dual variables). For the example defined by problem (4.6), The Hessians (second derivatives) of the objective and constraint functions are given by

$$\nabla^2 f(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 3x_2^2 \\ 0 & 3x_2^2 & 6x_1x_2 \end{bmatrix}, \quad \nabla^2 c_0(x) = \begin{bmatrix} -\cos(x_0) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$\nabla^2 c_1(x) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \nabla^2 c_2(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Scaling the constraint matrices by their corresponding Lagrange multipliers and summing, we get

$$H(x,\lambda) = \begin{bmatrix} -\lambda_0 \cos(x_0) + 2\lambda_1 & 0 & 0 \\ 0 & 2\lambda_1 & 3x_2^2 \\ 0 & 3x_2^2 & 6x_1x_2 \end{bmatrix}.$$

Since the Hessian matrix will always be a symmetric matrix, KNITRO only stores the nonzero elements corresponding to the upper triangular part (including the diagonal). In the example here, the number of nonzero elements in the upper triangular part of the Hessian matrix `nnzH` is 4. The KNITRO array `hess` stores the values of these elements, while the arrays `hessIndexRows` and `hessIndexCols` store the row and column indices respectively. The order in which these nonzero elements is stored is not important. If we store them column-wise, the arrays `hess`, `hessIndexRows` and `hessIndexCols` would look as follows.

```
hess[0] = -lambda[0]*cos(x[0]) + 2*lambda[1];
hessIndexRows[0] = 0;
hessIndexCols[0] = 0;

hess[1] = 2*lambda[1];
hessIndexRows[1] = 1;
hessIndexCols[1] = 1;

hess[2] = 3*x[2]*x[2];
hessIndexRows[2] = 1;
hessIndexCols[2] = 2;

hess[3] = 6*x[1]*x[2];
hessIndexRows[3] = 2;
hessIndexCols[3] = 2;
```

5 User options in KNITRO

5.1 Description of KNITRO user options

KNITRO 5.0 offers a number of user options, each option taking an integer or double precision value. Options are identified by a string name or an integer. Each integer identifier is associated with a C language macro defined in the file knitro.h. This section lists all user options, identified by the macro definition and string name. Sections 5.2 and 5.3 provide instructions on how to modify user options.

The integer valued options are:

`KTR_PARAM_ALG (algorithm):` Indicates which algorithm to use to solve the problem (see section 8).

- `0:` KNITRO will automatically try to choose the best algorithm based on the problem characteristics.
- `1:` KNITRO will use the Interior/Direct algorithm.
- `2:` KNITRO will use the Interior/CG algorithm.
- `3:` KNITRO will use the Active algorithm.

Default value: `0`

`KTR_PARAM_BARRULE (barrule):` Indicates which strategy to use for modifying the barrier parameter in the interior point code. Some strategies are only available for the Interior/Direct algorithm. (see section 8).

- `0 (AUTOMATIC):` KNITRO will automatically choose the rule for updating the barrier parameter.
- `1 (MONOTONE):` KNITRO will monotonically decrease the barrier parameter.
- `2 (ADAPTIVE):` KNITRO uses an adaptive rule based on the complementarity gap to determine the value of the barrier parameter at every iteration.
- `3 (PROBING):` KNITRO uses a probing (affine-scaling) step to dynamically determine the barrier parameter value at each iteration.
- `4 (DAMPMPC):` KNITRO uses a Mehrotra predictor-corrector type rule to determine the barrier parameter with safeguards on the corrector step.
- `5 (FULLMPC):` KNITRO uses a Mehrotra predictor-corrector type rule to determine the barrier parameter without safeguards on the corrector step.
- `6 (QUALITY):` KNITRO minimizes a *quality* function at each iteration to determine the barrier parameter.

Default value: `0`

NOTE: Only strategies 0-2 are available for the Interior/CG algorithm. All strategies are available for the Interior/Direct algorithm. Strategies 4 and 5 are typically recommended for linear programs or convex quadratic programs. Many problems benefit from a non-default setting of this option and it is recommended to experiment with all settings. In particular we recommend trying strategy 6 when using the Interior/Direct algorithm. This parameter is not applicable to the Active algorithm.

KTR_PARAM_NEWPOINT (newpoint): Specifies whether or not the new-point feature is enabled. If enabled, KNITRO returns control to the driver level with the return value from KTR_solve() = KTR_RC_NEWPOINT after a new solution estimate has been obtained and quantities have been updated (see section 9.7).

- 0: KNITRO will *not* return to the driver level after completing a successful iteration.
- 1: KNITRO will return to the driver level with the return value from KTR_solve() = KTR_RC_NEWPOINT after completing a successful iteration.

Default value: 0

KTR_PARAM_FEASIBLE (feasible): Indicates whether or not to use the feasible version of KNITRO.

- 0: Iterates may be infeasible.
- 1: Given an initial point which *sufficiently* satisfies all *inequality* constraints as defined by,

$$cl + tol \leq c(x) \leq cu - tol \tag{5.13}$$

(for $cl \neq cu$), the feasible version of KNITRO ensures that all subsequent solution estimates strictly satisfy the *inequality* constraints. However, the iterates may not be feasible with respect to the *equality* constraints. The tolerance $tol > 0$ in (5.13) for determining when the feasible mode is active is determined by the double precision parameter KTR_PARAM_FEASMODETOL described below. This tolerance (i.e. KTR_PARAM_FEASMODETOL) must be strictly positive. That is, in order to enter feasible mode, the point given to KNITRO must be strictly feasible with respect to the inequality constraints.

If the initial point is infeasible (or not sufficiently feasible according to (5.13)) with respect to the *inequality* constraints, then KNITRO will run the infeasible version until a point is obtained which sufficiently satisfies all the *inequality* constraints. At this point it will switch to feasible mode.

Default value: 0

NOTE: This option can only be used with the interior-point optimizers (i.e., when KTR_PARAM_ALG=1 or KTR_PARAM_ALG=2). See section 9.3 for more details.

KTR_PARAM_GRADOPT (gradopt): Specifies how to compute the gradients of the objective and constraint functions.

- 1: user will provide a routine for computing the exact gradients
- 2: gradients computed by forward finite-differences
- 3: gradients computed by central finite differences

Default value: 1

NOTE: It is highly recommended to provide exact gradients if at all possible as this greatly impacts the performance of the code. For more information on these options see section 9.1.

KTR_PARAM_HESSOPT (hessopt): Specifies how to compute the (approximate) Hessian of the Lagrangian.

1:	user will provide a routine for computing the exact Hessian
2:	KNITRO will compute a (dense) quasi-Newton BFGS Hessian
3:	KNITRO will compute a (dense) quasi-Newton SR1 Hessian
4:	KNITRO will compute Hessian-vector products using finite-differences
5:	user will provide a routine to compute the Hessian-vector products
6:	KNITRO will compute a limited-memory quasi-Newton BFGS Hessian

Default value: 1

NOTE: If exact Hessians (or exact Hessian-vector products) cannot be provided by the user but exact gradients are provided and are not too expensive to compute, option 4 above is typically recommended. The finite-difference Hessian-vector option is comparable in terms of robustness to the exact Hessian option (*assuming exact gradients are provided*) and typically not too much slower in terms of time if gradient evaluations are not the dominant cost.

However, if exact gradients cannot be provided (i.e. finite-differences are used for the first derivatives), or gradient evaluations are expensive, it is recommended to use one of the quasi-Newton options, in the event that the exact Hessian is not available. Options 2 and 3 are only recommended for small problems ($n < 1000$) since they require working with a dense Hessian approximation. Option 6 should be used in the large-scale case.

NOTE: Options KTR_PARAM_HESSOPT=4 and KTR_PARAM_HESSOPT=5 are not available when KTR_PARAM_ALG=1. See section 9.2 for more detail on second derivative options.

KTR_PARAM_LMSIZE (lmsize): Specifies the number of limited memory pairs stored when approximating the Hessian using the limited-memory quasi-Newton BFGS option. The value must be between 1 and 100 and is only used when KTR_PARAM_HESSOPT=6. Larger values may give a more accurate, but more expensive, Hessian approximation. Smaller values may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter. See section 9.2 for more details.

Default value: 10

KTR_PARAM_HONORBNDS (honorbnds): Indicates whether or not to enforce satisfaction of simple variable bounds throughout the optimization (see section 9.4).

0:	KNITRO does not require that the bounds on the variables be satisfied at intermediate iterates.
1:	KNITRO enforces that the initial point and all subsequent solution estimates satisfy the bounds on the variables.

Default value: 0

KTR_PARAM_INITPT (initpt): Indicates whether an initial point strategy is used.

`0:` No initial point strategy is employed.

`1:` Initial values for the variables are computed. This option may be recommended when an initial point is not provided by the user, as is typically the case in linear and quadratic programming problems.

Default value: `0`

`KTR_PARAM_LPSOLVER (lpsolver):` Indicates which linear programming simplex solver the KNITRO active-set algorithm uses to solve the LP subproblems.

`1:` KNITRO uses an internal LP solver.

`2:` KNITRO uses ILOG-CPLEX provided the user has a valid CPLEX license. The CPLEX shared object library or dll must reside in the current working directory or a directory specified in the library load path in order to be run-time loadable. If this option is selected, KNITRO will look for (in order): CPLEX 10.0, CPLEX 9.1, CPLEX 9.0, or CPLEX 8.0. To specify a particular CPLEX library, set it as the character type user option `"cplexlibname"`. For example, to specifically load the library `cplex90.dll`, call the following (and be sure to check the return status):

```
KTR_set_char_param_by_name(kc,"cplexlibname","cplex90.dll");
```

Default value: `1`

`KTR_PARAM_MULTISTART (multistart):` Indicates whether KNITRO will solve from multiple start points to find a better local minimum. See section 9.6 for details.

`0:` KNITRO solves from a single initial point.

`1:` KNITRO solves using multiple start points.

Default value: `0`

`KTR_PARAM_MSMAXSOLVES (ms_maxsolves):` Specifies how many start points to try in multistart. This option is only valid if `KTR_PARAM_MULTISTART=1`.

Default value: `1`

`KTR_PARAM_MAXCGIT (maxcgit):` Determines the maximum allowable number of inner conjugate gradient (CG) iterations per KNITRO minor iteration.

`0:` KNITRO automatically determines an upper bound on the number of allowable CG iterations based on the problem size.

n`:` At most n CG iterations may be performed during one KNITRO minor iteration, where $n > 0$.

Default value: `0`

KTR_PARAM_MAXCROSSIT (maxcrossit): Specifies the maximum number of crossover iterations before termination. When running one of the interior-point algorithms in KNITRO, if this value is positive, it will switch to the Active algorithm near the solution, and perform at most KTR_PARAM_MAXCROSSIT iterations of the Active algorithm to try to get a more exact solution. If this value is 0 or negative, no Active crossover iterations will be performed.

If crossover is unable to improve the approximate interior-point solution, then it will restore the interior-point solution. In some cases (especially on large-scale problems or difficult degenerate problems) the cost of the crossover procedure may be significant – for this reason, the crossover procedure is disabled by default. However, in many cases the additional cost of performing crossover is not significant and you may wish to enable this feature to obtain a more accurate solution. See section 9.5 for more details on the crossover procedure.

Default value: 0

KTR_PARAM_MAXIT (maxit): Specifies the maximum number of major iterations before termination.

Default value: 10000

KTR_PARAM_OUTLEV (outlev): Controls the level of output.

0:	printing of all output is suppressed
1:	print only summary information
2:	print information every 10 major iterations, where a major iteration is defined by a new solution estimate
3:	print information at each major iteration
4:	print information at each major and minor iteration, where a minor iteration is defined by a trial iterate
5:	print all the above, and the values of the solution vector x
6:	print all the above, and the values of the constraints c at x and the Lagrange multipliers lambda

Default value: 2

KTR_PARAM_OUTMODE (outmode): Specifies where to direct the output.

0:	output is directed to standard out (e.g., screen)
1:	output is sent to a file named knitro.log
2:	output is directed to both the screen and file knitro.log

Default value: 0

KTR_PARAM_SCALE (scale): Performs a scaling of the objective and constraint functions based on their values at the initial point. If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.

0:	No scaling is performed.
1:	The objective function and constraints may be scaled.

Default value: 1

KTR_PARAM_SHIFTINIT (shiftinit): Determines whether or not the interior-point algorithm in KNITRO shifts the given initial point to satisfy bounds on the variables.

- 0: KNITRO will not shift the given initial point to satisfy the variable bounds before starting the optimization.
- 1: KNITRO will shift the given initial point.

Default value: 1

KTR_PARAM_SOC (soc): Indicates whether or not to use the second order correction (SOC) option. A second order correction may be beneficial for problems with highly nonlinear constraints.

- 0: No second order correction steps are attempted.
- 1: Second order correction steps may be attempted on some iterations.
- 2: Second order correction steps are always attempted if the original step is rejected and there are nonlinear constraints.

Default value: 1

KTR_PARAM_DEBUG (debug): Controls the level of debugging output. Debugging output can slow execution of KNITRO and should not be used in a production setting. All debugging output is suppressed if option outlev equals zero.

- 0: No debugging output.
- 1: Print algorithm information to .log output files.
- 2: Print program execution information.

Default value: 0

The double precision valued options are:

KTR_PARAM_DELTA (delta): Specifies the initial trust region radius scaling factor used to determine the initial trust region size.

Default value: 1.0e0

KTR_PARAM_FEASMODETOL (feasmodetol): Specifies the tolerance in (5.13) by which the iterate must be feasible with respect to the inequality constraints before the feasible mode becomes active. This option is only relevant when KTR_PARAM_FEASIBLE=1.

Default value: 1.0e-4

KTR_PARAM_FEASTOL (feastol): Specifies the final relative stopping tolerance for the feasibility error. Smaller values of KTR_PARAM_FEASTOL result in a higher degree of accuracy in the solution with respect to feasibility.

Default value: 1.0e-6

KTR_PARAM_FEASTOLABS (feastolabs): Specifies the final absolute stopping tolerance for the feasibility error. Smaller values of KTR_PARAM_FEASTOLABS result in a higher degree of accuracy in the solution with respect to feasibility.

Default value: 0.0e0

NOTE: For more information on the stopping test used in KNITRO see section 6.

KTR_PARAM_MAXTIMECPU (maxtime_cpu): Specifies, in seconds, the maximum allowable CPU time before termination.

Default value: 1.0e8

KTR_PARAM_MAXTIMEREAL (maxtime_real): Specifies, in seconds, the maximum allowable real time before termination.

Default value: 1.0e8

KTR_PARAM_MU (mu): specifies the initial barrier parameter value for the interior-point algorithms.

Default value: 1.0e-1

KTR_PARAM_OBJRANGE (objrange): Determines the allowable range of values for the objective function for determining unboundedness. If the magnitude of the objective function is greater than KTR_PARAM_OBJRANGE and the iterate is feasible, then the problem is determined to be unbounded.

Default value: 1.0e20

KTR_PARAM_OPTTOL (opttol): Specifies the final relative stopping tolerance for the KKT (optimality) error. Smaller values of KTR_PARAM_OPTTOL result in a higher degree of accuracy in the solution with respect to optimality.

Default value: 1.0e-6

KTR_PARAM_OPTTOLABS (opttolabs): Specifies the final absolute stopping tolerance for the KKT (optimality) error. Smaller values of KTR_PARAM_OPTTOLABS result in a higher degree of accuracy in the solution with respect to optimality.

Default value: 0.0e0

NOTE: For more information on the stopping test used in KNITRO see section 6.

KTR_PARAM_PIVOT (pivot): Specifies the initial pivot threshold used in the factorization routine. The value should be in the range [0 0.5] with higher values resulting in more pivoting (more stable factorization). Values less than 0 will be set to 0 and values larger than 0.5 will be set to 0.5. If pivot is non-positive initially no pivoting will be performed. Smaller values may improve the speed of the code but higher values are recommended for more stability (for example, if the problem appears to be very ill-conditioned).

Default value: 1.0e-8

KTR_PARAM_XTOL (xtol): The optimization will terminate when the relative change in the solution estimate is less than KTR_PARAM_XTOL. If using an interior-point algorithm and the barrier parameter is still large, KNITRO will first try decreasing the barrier parameter before terminating.

Default value: 1.0e-15

5.2 The KNITRO options file

The KNITRO options file allows the user to easily change certain parameters without needing to recompile the code when using the KNITRO callable library. (This file has no effect when using the AMPL interface to KNITRO.)

Options are set by specifying a keyword and a corresponding value on a line in the options file. Lines that begin with a # character are treated as comments and blank lines are ignored. For example, to set the maximum allowable number of iterations to 500, one could use the following options file. In this example, let's call the options file `knitro.opt` (although any name will do):

```
# KNITRO Options file
maxit        500
```

In order for the options file to be read by KNITRO the following function call must be specified in the driver:

```
int KTR_load_param_file(KTR_context *kc, char const *filename)
```

Example:

```
status = KTR_load_param_file(kc,"knitro.opt");
```

Likewise, the options used in a given optimization may be written to a file called `knitro.opt` through the function call:

```
int KTR_save_param_file(KTR_context *kc, char const *filename)
```

Example:

```
status = KTR_save_param_file(kc,"knitro.opt");
```

A sample options file knitro.opt is provided for convenience and can be found in the `C` and `Fortran` directories. Note that this file is only provided as a sample: it is not read by KNITRO (unless the user specifies the function call for reading this file).

Most user options can be specified with either a numeric value or a string value. The individual user options and their possible numeric values are described in section 5.1. Optional string values for many of the options are indicated in the example knitro.opt file provided with the distribution.

5.3 Setting options through function calls

The functions for setting user parameters have the form:

```
int KTR_set_int_param(KTR_context *kc, int param, int value)
```

for setting integer valued parameters or

```
int KTR_set_double_param(KTR_context *kc, int param, double value)
```

for setting double precision valued parameters.

Example:

The `Interior/CG` algorithm can be chosen through the following function call.

```
status = KTR_set_int_param(kc,KTR_PARAM_ALG, 2);
```

Similarly, the optimality tolerance can be set through the function call:

```
status = KTR_set_double_param(kc,KTR_PARAM_OPTTOL, 1.0e-8);
```

NOTE: User parameters should only be set at the very beginning of the optimization before the call to `KTR_solve()` and should not be modified at all during the course of the optimization.

6 KNITRO termination test and optimality

The first-order conditions for identifying a locally optimal solution of the problem (1.1) are:

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) + \sum_{i\in\mathcal{E}} \lambda_i \nabla h_i(x) + \sum_{i\in I} \lambda_i \nabla g_i(x) = 0 \tag{6.14}$$

$$\lambda_i g_i(x) = 0, \quad i \in I \tag{6.15}$$

$$h_i(x) = 0, \quad i \in \mathcal{E} \tag{6.16}$$

$$g_i(x) \leq 0, \quad i \in I \tag{6.17}$$

$$\lambda_i \geq 0, \quad i \in I \tag{6.18}$$

where $\mathcal{E}$ and I represent the sets of indices corresponding to the equality constraints and inequality constraints respectively, and λ_i is the Lagrange multiplier corresponding to constraint i. In KNITRO we define the feasibility error (`Feas err`) at a point x^k to be the maximum violation of the constraints (6.16), (6.17), i.e.,

$$\text{Feas err} = \max_{i\in\mathcal{E}\cup I} (0, |h_i(x^k)|, g_i(x^k)), \tag{6.19}$$

while the optimality error (`Opt err`) is defined as the maximum violation of the first two conditions (6.14), (6.15),

$$\text{Opt err} = \max_{i\in I} (\|\nabla_x \mathcal{L}(x^k,\lambda^k)\|_\infty, \min(|\lambda_i^k g_i(x^k)|, |\lambda_i^k|, |g_i(x^k)|)). \tag{6.20}$$

The last optimality condition (6.18) is enforced explicitly throughout the optimization. In order to take into account problem scaling in the termination test, the following scaling factors are defined

$$\tau_1 = \max(1, |h_i(x^0)|, g_i(x^0)), \tag{6.21}$$

$$\tau_2 = \max(1, \|\nabla f(x^k)\|_\infty), \tag{6.22}$$

where x^0 represents the initial point.

For unconstrained problems, the scaling (6.22) is not effective since $\|\nabla f(x^k)\|_\infty \to 0$ as a solution is approached. Therefore, for unconstrained problems only, the following scaling is used in the termination test

$$\tau_2 = \max(1, \min(|f(x^k)|, \|\nabla f(x^0)\|_\infty)), \tag{6.23}$$

in place of (6.22).

KNITRO stops and declares `LOCALLY OPTIMAL SOLUTION FOUND` if the following stopping conditions are satisfied:

$$\text{Feas err} \leq \max(\tau_1 * \texttt{KTR_PARAM_FEASTOL}, \texttt{KTR_PARAM_FEASTOLABS}) \tag{6.24}$$

$$\text{Opt err} \leq \max(\tau_2 * \texttt{KTR_PARAM_OPTTOL}, \texttt{KTR_PARAM_OPTTOLABS}) \tag{6.25}$$

where `KTR_PARAM_FEASTOL`, `KTR_PARAM_OPTTOL`, `KTR_PARAM_FEASTOLABS` and `KTR_PARAM_OPTTOLABS` are user-defined options (see section 5).

This stopping test is designed to give the user much flexibility in deciding when the solution returned by KNITRO is accurate enough. One can use a purely scaled stopping test (which is the recommended default option) by setting `KTR_PARAM_FEASTOLABS` and `KTR_PARAM_OPTTOLABS` equal to `0.0e0`. Likewise, an absolute stopping test can be enforced by setting `KTR_PARAM_FEASTOL` and `KTR_PARAM_OPTTOL` equal to `0.0e0`.

Unbounded problems

Since by default, KNITRO uses a relative/scaled stopping test it is possible for the optimality conditions to be satisfied within the tolerances given by (6.24)-(6.25) for an unbounded problem. For example, if $\tau_2 \rightarrow \infty$ while the optimality error (6.20) stays bounded, condition (6.25) will eventually be satisfied for some `KTR_PARAM_OPTTOL>0`. If you suspect that your problem may be unbounded, using an absolute stopping test will allow KNITRO to detect this.

7 KNITRO output and solution information

This section provides information on understanding the KNITRO output and accessing solution information.

7.1 Understanding KNITRO output

If KTR_PARAM_OUTLEV=0 then all printing of output is suppressed. If KTR_PARAM_OUTLEV>0 then KNITRO will print information about the solution of your optimization problem either to standard output, e.g., the screen (KTR_PARAM_OUTMODE=0), to a file named knitro.log (KTR_PARAM_OUTMODE=1) or to both (KTR_PARAM_OUTMODE=2).

For various levels of output printing, the following information is given. In the example output below we use the example provided in examples/C/problemHS15.c

Nondefault Options:

If KTR_PARAM_OUTLEV>0, then KNITRO first prints the banner displaying the Ziena license type and version of KNITRO being used. It then lists all user options which are different from their default values (see section 5 for the default user option settings). If nothing is listed in this section then all user options are set to their default values. If the automatic algorithm selection option algorithm=0 is being used, it also prints which algorithm was selected.

```
=====================================
       Commercial Ziena License
             KNITRO 5.0.0
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
=====================================

outlev:       6
Automatic algorithm selection: Interior/Direct
```

In the example above, it is indicated that we are using a more verbose output level outlev=6 instead of the default value outlev=2 and that the automatic algorithm selection option chose the Interior/Direct algorithm.

Problem Characteristics:

If KTR_PARAM_OUTLEV>0, KNITRO next prints a summary description of the problem characteristics including the number and type of variables and constraints and the number of nonzero elements in the Jacobian matrix and Hessian matrix (if providing the exact Hessian).

```
Problem Characteristics
-----------------------
Number of variables:                    2
    bounded below:                      0
    bounded above:                      1
    bounded below and above:            0
    fixed:                              0
```

```
    free:                                1
Number of constraints:                   2
    linear equalities:                   0
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              2
    range:                               0
Number of nonzeros in Jacobian:          4
Number of nonzeros in Hessian:           3
```

Iteration Information:

Next, if KTR_PARAM_OUTLEV$\geq$ 2, KNITRO will print columns of data reflecting some detailed information about individual iterations during the solution process. A major iteration, in the context of KNITRO, is defined as a step which generates a new solution estimate (i.e., a successful step). A minor iteration is one which generates a trial step (which may either be accepted or rejected).

If KTR_PARAM_OUTLEV$=$ 2, this data is printed every 10 major iterations (and on the final iteration). If KTR_PARAM_OUTLEV$=$ 3, this data is printed every major iteration. If KTR_PARAM_OUTLEV$\geq$ 4, this information is printed for every major and minor iteration.

Below is a description of the values contained under each column header and an example of the iteration output for `outlev`=6.

`Iter:` Iteration number (major/minor).

`Res:` The step result. The values in this column indicate whether or not the step attempted during the iteration was accepted (`Acc`) or rejected (`Rej`) by the merit function. If the step was rejected, the solution estimate was not updated. (This information is only printed if KTR_PARAM_OUTLEV>3).

`Objective:` Gives the value of the objective function at the trial iterate.

`Feas err:` Gives a measure of the feasibility violation at the trial iterate (see section 6).

`Opt Err:` Gives a measure of the violation of the Karush-Kuhn-Tucker (KKT) (first-order) optimality conditions (not including feasibility) (see section 6).

`||Step||:` The 2-norm length of the step (i.e., the distance between the trial iterate and the old iterate).

`CG its:` The number of Projected Conjugate Gradient (CG) iterations required to compute the step.

```
 Iter(maj/min)  Res   Objective      Feas err    Opt err     ||Step||    CG its
--------------- ---  -------------  ----------  ----------  ----------  -------
      0/      0 ---   9.090000e+02   3.000e+00
      1/      1 Acc   7.989784e+02   2.878e+00   9.096e+01   6.566e-02        0
      2/      2 Acc   4.232342e+02   2.554e+00   5.828e+01   2.356e-01        0
      3/      3 Acc   1.457686e+01   9.532e-01   3.088e+00   1.909e+00        0
              4 Rej   3.227542e+02   9.532e-01   3.088e+00   1.483e+01        1
              5 Rej   1.803608e+03   9.532e-01   3.088e+00   7.330e+00        1
              6 Rej   1.176121e+03   9.532e-01   3.088e+00   3.576e+00        1
              7 Rej   4.249636e+02   9.532e-01   3.088e+00   1.698e+00        1
```

```
  4/      8 Acc   1.235269e+02   7.860e-01   3.818e+00   7.601e-01        1
  5/      9 Acc   3.993788e+02   3.022e-02   1.795e+01   1.186e+00        0
  6/     10 Acc   3.924231e+02   2.924e-02   1.038e+01   1.856e-02        0
  7/     11 Acc   3.158787e+02   0.000e+00   6.905e-02   2.373e-01        0
  8/     12 Acc   3.075530e+02   0.000e+00   6.888e-03   2.255e-02        0
  9/     13 Acc   3.065107e+02   0.000e+00   6.397e-05   2.699e-03        0
 10/     14 Acc   3.065001e+02   0.000e+00   4.457e-07   2.714e-05        0
```

Termination Message:

At the end of the run, if KTR_PARAM_OUTLEV>0, a termination message is printed indicating whether or not the optimal solution was found and if not, why the code terminated. The termination message typically starts with EXIT:. If KNITRO was successful in satisfying the termination test (see section 6), the message will look as follows:

```
EXIT: LOCALLY OPTIMAL SOLUTION FOUND.
```

See the appendix for a list of possible termination messages and a description of their meaning and corresponding return value.

Final Statistics:

Following the termination message, if KTR_PARAM_OUTLEV>0, a summary of some final statistics on the run are printed. Both relative and absolute error values are printed.

```
Final Statistics
----------------
Final objective value               =   3.06500096351765e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   4.46e-07 / 3.06e-08
# of iterations (major / minor)     =         10 /       14
# of function evaluations           =         15
# of gradient evaluations           =         11
# of Hessian evaluations            =         10
Total program time (secs)           =       0.00136 (     0.000 CPU time)
```

Solution Vector/Constraints:

If KTR_PARAM_OUTLEV ≥ 5, the values of the solution vector are printed after the final statistics. If KTR_PARAM_OUTLEV=6, the final constraint values are also printed before the solution vector, and the values of the Lagrange multipliers (or dual variables) are printed next to their corresponding constraint or bound.

```
Constraint Vector                   Lagrange Multipliers
-----------------                   --------------------
c[       0] =   1.00000006873e+00,   lambda[       0] =  -7.00000062964e+02
c[       1] =   4.50000096310e+00,   lambda[       1] =  -1.07240081095e-05

Solution Vector
---------------
x[       0] =   4.99999972449e-01,   lambda[       2] =   7.27764067199e+01
x[       1] =   2.00000024766e+00,   lambda[       3] =   0.00000000000e+00

=============================================================================
```

In addition to the information above, KNITRO can produce some additional information which may be useful in debugging or analyzing performance. If KTR_PARAM_OUTLEV>0 and KTR_PARAM_DEBUG=1, then multiple .log files are created which contain detailed information on performance. If KTR_PARAM_OUTLEV>0 and KTR_PARAM_DEBUG=2, then KNITRO prints additional program execution information. The information produced by KTR_PARAM_DEBUG is primarily intended for debugging by developers and should not be used in a production setting.

7.2 Accessing solution information

Important solution information from KNITRO is either made available as output from the call to KTR_solve() or can be retrieved through special function calls.

The KTR_solve() function (see section 4) returns the final value of the objective function in obj, the final (primal) solution vector in the array x and the final values of the Lagrange multipliers (or dual variables) in the array lambda. The solution status code is given by the return value from KTR_solve().

In addition, information related to the final statistics can be retrieved through the following function calls:

```
int KTR_get_number_FC_evals (const KTR_context_ptr  kc);
```

This function call returns the number of function evaluations requested by KTR_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_GA_evals (const KTR_context_ptr  kc);
```

This function call returns the number of gradient evaluations requested by KTR_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_H_evals (const KTR_context_ptr  kc);
```

This function call returns the number of Hessian evaluations requested by KTR_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_HV_evals (const KTR_context_ptr  kc);
```

This function call returns the number of Hessian-vector products requested by KTR_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_major_iters (const KTR_context_ptr  kc);
```

This function returns the number of major iterations made by KTR_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_minor_iters (const KTR_context_ptr  kc);
```

This function returns the number of minor iterations made by KTR_solve(). It returns a negative number if there is a problem with kc.

```
double KTR_get_abs_feas_error (const KTR_context_ptr  kc);
```

This function returns the absolute feasibility error at the solution. See 6 for a detailed definition of this quantity. It returns a negative number if there is a problem with kc.

```
double KTR_get_abs_opt_error (const KTR_context_ptr  kc);
```

This function returns the absolute optimality error at the solution. See 6 for a detailed definition of this quantity. It returns a negative number if there is a problem with kc.

8 Algorithm Options

8.1 Automatic

By default, KNITRO will automatically try to choose the best optimizer for the given problem based on the problem characteristics.

8.2 Interior/Direct

If the Hessian of the Lagrangian is ill-conditioned or the problem does not have a large-dense Hessian, it may be advisable to compute a step by directly factoring the KKT (primal-dual) matrix rather than using an iterative approach to solve this system. KNITRO offers the Interior/Direct optimizer which allows the algorithm to take Newton-like steps using a direct factorization approach by setting `KTR_PARAM_ALG=1`. This option will try to take a direct step at each iteration and will only fall back on the iterative step if the direct step is suspected to be of poor quality, or if negative curvature is detected.

Using the Interior/Direct optimizer may result in substantial improvements over Interior/CG when the problem is ill-conditioned (as evidenced by Interior/CG taking a large number of Conjugate Gradient iterations). We encourage the user to try both options as it is difficult to predict in advance which one will be more effective on a given problem.

NOTE: Since the Interior/Direct algorithm in KNITRO requires the explicit storage of a Hessian matrix, this version can only be used with Hessian options, `KTR_PARAM_HESSOPT=1, 2, 3` or `6`. It may not be used with Hessian options, `KTR_PARAM_HESSOPT=4` or `5`, which only provide Hessian-vector products. The Interior/Direct optimizer may be used with the `feasible` option.

8.3 Interior/CG

Since KNITRO was designed with the idea of solving large problems, the Interior/CG optimizer in KNITRO offers an iterative Conjugate Gradient approach to compute the step at each iteration. This approach has proven to be efficient in most cases and allows KNITRO to handle problems with large, dense Hessians, since it does not require factorization of the Hessian matrix. The Interior/CG algorithm can be chosen by setting `KTR_PARAM_ALG=2`. It can use any of the Hessian options as well as the `feasible` option.

8.4 Active

KNITRO also features an active-set Sequential Linear-Quadratic Programing (SLQP) optimizer. This optimizer is particularly advantageous when "warm starting" (i.e., when the user can provide a good initial solution estimate, for example, when solving a sequence of closely related problems). This algorithm is also best at rapid detection of infeasible problems. The Active algorithm is efficient and robust for small and medium-scale problems, but is typically less efficient that both the Interior/Direct and Interior/CG algorithms on large-scale problems (problems with many thousands of variables and constraints). The Active algorithm can be chosen by setting `KTR_PARAM_ALG=3`. It can use any of the Hessian options.

NOTE: The `feasible` option (see section 9.3) is not available for use with the Active optimizer.

We strongly encourage you to experiment with all algorithm options as it is difficult to predict which one will work best on any particular problem.

9 Other KNITRO special features

This section describes in more detail some of the most important features of KNITRO. It provides some guidance on which features to use so that KNITRO runs most efficiently for the problem at hand.

9.1 First derivative and gradient check options

The default version of KNITRO assumes that the user can provide exact first derivatives to compute the objective function gradient and constraint gradients. It is *highly* recommended that the user provide exact first derivatives if at all possible, since using first derivative approximations may seriously degrade the performance of the code and the likelihood of converging to a solution. However, if this is not possible the following first derivative approximation options may be used.

Forward finite-differences
This option uses a forward finite-difference approximation of the objective and constraint gradients. The cost of computing this approximation is n function evaluations where n is the number of variables. The option is invoked by choosing user option `KTR_PARAM_GRADOPT=2` (see section 5).

Centered finite-differences
This option uses a centered finite-difference approximation of the objective and constraint gradients. The cost of computing this approximation is $2n$ function evaluations where n is the number of variables. The option is invoked by choosing user option `KTR_PARAM_GRADOPT=3` (see section 5). The centered finite-difference approximation is often more accurate than the forward finite-difference approximation; however, it is more expensive to compute if the cost of evaluating a function is high.

Gradient Checks
If the user supplies a routine for computing exact gradients, KNITRO can easily check them against finite-difference gradient approximations. To do this, modify the application by replacing `KTR_solve()` with `KTR_check_first_ders()`, and run the application. KNITRO will call the user routine for exact gradients, compute finite-difference approximations, and print any differences that exceed a given threshold. KNITRO also checks that the sparse constraint Jacobian has all nonzero elements defined. The check can be made with forward or centered differences. A sample driver is provided in examples/C/checkDersExample.c. Small differences between exact and finite-difference approximations are to be expected (see comments in examples/C/checkDersExample.c). It is best to check the gradient at different points, and to avoid points where partial derivatives happen to equal zero.

9.2 Second derivative options

The default version of KNITRO assumes that the user can provide exact second derivatives to compute the Hessian of the Lagrangian function. If the user is able to do so and the cost of computing the second derivatives is not overly expensive, it is highly recommended to provide exact second derivatives. However, KNITRO also offers other options which are described in detail below.

(Dense) Quasi-Newton BFGS
The quasi-Newton BFGS option uses gradient information to compute a symmetric, *positive-definite* approximation to the Hessian matrix. Typically this method requires more iterations to converge than the exact Hessian version. However, since it is only computing gradients rather than Hessians, this approach may be

more efficient in some cases. This option stores a *dense* quasi-Newton Hessian approximation so it is only recommended for small to medium problems ($n < 1000$). The quasi-Newton BFGS option can be chosen by setting options value KTR_PARAM_HESSOPT=2.

(Dense) Quasi-Newton SR1
As with the BFGS approach, the quasi-Newton SR1 approach builds an approximate Hessian using gradient information. However, unlike the BFGS approximation, the SR1 Hessian approximation is not restricted to be positive-definite. Therefore the quasi-Newton SR1 approximation may be a better approach, compared to the BFGS method, if there is a lot of negative curvature in the problem since it may be able to maintain a better approximation to the true Hessian in this case. The quasi-Newton SR1 approximation maintains a *dense* Hessian approximation and so is only recommended for small to medium problems ($n < 1000$). The quasi-Newton SR1 option can be chosen by setting options value KTR_PARAM_HESSOPT=3.

Finite-difference Hessian-vector product option
If the problem is large and gradient evaluations are not the dominate cost, then KNITRO can internally compute Hessian-vector products using finite-differences. Each Hessian-vector product in this case requires one additional gradient evaluation. This option can be chosen by setting options value KTR_PARAM_HESSOPT=4. This option is generally only recommended if the exact gradients are provided.

NOTE: This option may not be used when KTR_PARAM_ALG=1.

Exact Hessian-vector products
In some cases the user may have a large, dense Hessian which makes it impractical to store or work with the Hessian directly, but the user may be able to provide a routine for evaluating exact Hessian-vector products. KNITRO provides the user with this option. If this option is selected, the user can provide a routine which given a vector v stored in `hessVector`, computes the Hessian-vector product, Hv, and stores the result in `hessVector`. This option can be chosen by setting options value KTR_PARAM_HESSOPT=5.

NOTE: This option may not be used when KTR_PARAM_ALG=1.

Limited-memory Quasi-Newton BFGS
The limited-memory quasi-Newton BFGS option is similar to the dense quasi-Newton BFGS option described above. However, it is better suited for large-scale problems since, instead of storing a dense Hessian approximation, it only stores a limited number of gradient vectors used to approximate the Hessian. The number of gradient vectors used to approximate the Hessian is controlled by user option KTR_PARAM_LMSIZE which must be between 1 and 100 (10 is the default).

A larger value of KTR_PARAM_LMSIZE may result in a more accurate, but also more expensive, Hessian approximation. A smaller value may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter.

In general, the limited-memory BFGS option requires more iterations to converge than the dense quasi-Newton BFGS approach, but will be much more efficient on large-scale problems. This option can be chosen by setting options value KTR_PARAM_HESSOPT=6.

9.3 Feasible version

KNITRO offers the user the option of forcing intermediate iterates to stay feasible with respect to the *inequality* constraints (it does not enforce feasibility with respect to the *equality* constraints however). Given an initial

point which is *sufficiently* feasible with respect to all inequality constraints and selecting KTR_PARAM_FEASIBLE = 1, forces all the iterates to strictly satisfy the inequality constraints throughout the solution process. For the feasible mode to become active the iterate x must satisfy

$$cl + tol \leq c(x) \leq cu - tol \tag{9.26}$$

for *all* inequality constraints (i.e., for $cl \neq cu$). The tolerance $tol > 0$ by which an iterate must be strictly feasible for entering the feasible mode is determined by the parameter KTR_PARAM_FEASMODETOL which is 1.0e-4 by default. If the initial point does not satisfy (9.26) then the default infeasible version of KNITRO will run until it obtains a point which is sufficiently feasible with respect to all the inequality constraints. At this point it will switch to the feasible version of KNITRO and all subsequent iterates will be forced to satisfy the inequality constraints.

For a detailed description of the feasible version of KNITRO see [5].

NOTE: This option can only be used with the interior-point optimizers (i.e., when KTR_PARAM_ALG=1 or KTR_PARAM_ALG=2).

9.4 Honor Bounds

By default KNITRO does not enforce that the simple bounds on the variables be satisfied throughout the optimization process. Rather, satisfaction of these bounds is only enforced at the solution. In some applications, however, the user may want to enforce that the initial point and all intermediate iterates satisfy the bounds $bl \leq x \leq bu$. For instance, if the objective function or a nonlinear constraint function is undefined at points outside the bounds, then the bounds should be enforced. To enforce, set KTR_PARAM_HONORBNDS=1.

9.5 Crossover

Interior point (or barrier) methods are a powerful tool for solving large-scale optimization problems. However, one drawback of these methods, in contrast to active-set methods, is that they do not always provide a clear picture of which constraints are active at the solution and in general they return a less exact solution and less exact sensitivity information. For this reason, KNITRO offers a *crossover* feature in which the interior-point method switches to the active-set method at the interior-point solution estimate, in order to try to "clean up" the solution and provide more exact sensitivity and active-set information.

The crossover procedure is controlled by the KTR_PARAM_MAXCROSSIT user option. If this parameter is greater than 0, then KNITRO will attempt to perform KTR_PARAM_MAXCROSSIT active-set crossover iterations after the interior-point method has finished, to see if it can provide a more exact solution. This can be viewed as a form of post-processing. If KTR_PARAM_MAXCROSSIT<=0, then no crossover iterations are attempted. By default, no crossover iterations are performed.

The crossover procedure will not always succeed in obtaining a more exact solution compared with the interior-point solution. If crossover is unable to improve the solution within KTR_PARAM_MAXCROSSIT crossover iterations, then it will restore the interior-point solution estimate and terminate. If KTR_PARAM_OUTLEV>1, KNITRO will print a message indicating that it was unable to improve the solution. For example, if KTR_PARAM_MAXCROSSIT=3, and the crossover procedure did not succeed within 3 iterations, the message will read:

```
Crossover mode unable to improve solution within 3 iterations.
```

In this case, you may want to increase the value of KTR_PARAM_MAXCROSSIT and try again. If it appears that the crossover procedure will not succeed, no matter how many iterations are tried, then a message of the form

```
Crossover mode unable to improve solution.
```

will be printed.

The extra cost of performing crossover is problem dependent. In most small or medium scale problems, the crossover cost should be a small fraction of the total solve cost. In these cases it may be worth using the crossover procedure to obtain a more exact solution. On some large scale or difficult degenerate problems, however, the cost of performing crossover may be significant. It is recommended to experiment with this option to see whether the improvement in the exactness of the solution is worth the additional cost.

9.6 Multi-start

Nonlinear optimization problems (1.1) are often nonconvex due to the objective function, constraint functions, or both. When this is true, there may be many points that satisfy the local optimality conditions described in section 6. Default KNITRO behavior is to return the first locally optimal point found. KNITRO 5.0 offers a simple *multi-start* feature that searches for a better optimal point by restarting KNITRO from different initial points. The feature is enabled by setting KTR_PARAM_MULTISTART=1.

The multi-start procedure generates new start points by randomly selecting x components that satisfy the variable bounds. The number of start points to try is specified with the option KTR_PARAM_MSMAXSOLVES. KNITRO finds a local optimum from each start point using the same problem definition and user options. The solution returned from KTR_solve() is the local optimum with the best objective function value. If KTR_PARAM_OUTLEV is greater than 3, then KNITRO prints details of each local optimum.

The multi-start option is convenient for conducting a simple search for a better solution point. It can also save execution and memory overhead by internally managing the solve process from successive start points.

In most cases the user would like to obtain the *global optimum* to (1.1); that is, the local optimum with the very best objective function value. KNITRO cannot guarantee that mult-start will find the global optimum. The probability of finding a better point improves if more start points are tried, but so does the execution time. Search time can be improved if the variable bounds are made as tight as possible. Minimally, *all* variables need to be given finite upper and lower bounds.

9.7 Reverse communication mode for invoking KNITRO

The reverse communication mode of KNITRO returns control to the user at the driver level whenever a function, gradient, or Hessian evaluation is needed, thus providing maximum freedom in embedding the KNITRO solver into an application. In addition, this feature makes it easy for users to monitor or stop the progress of the algorithm after each iteration, based on whatever criteria the user desires.

If the return value from KTR_solve() is 0 or negative, the optimization is finished (0 indicates successful completion, whereas a negative return value indicates unsuccessful completion). Otherwise, if the return value is positive, KNITRO requires that some task be performed by the user at the driver level before re-entering KTR_solve(). Referring to the optimization problem formulation given in (4.4), the action to take for possible positive return values are:

KTR_RC_EVALFC **(1)** Evaluate functions $f(x)$ and $c(x)$ and re-enter KTR_solve().

KTR_RC_EVALGA **(2)** Evaluate gradient $\nabla f(x)$ and the constraint Jacobian and re-enter KTR_solve().

KTR_RC_EVALH **(3)** Evaluate the Hessian $H(x,\lambda)$ and re-enter `KTR_solve()`.

KTR_RC_EVALHV **(7)** Evaluate the Hessian $H(x,\lambda)$ times a vector and re-enter `KTR_solve()`.

KTR_RC_NEWPOINT **(6)** KNITRO has just computed a new solution estimate, and the function and gradient values are up-to-date. The user may provide routines to perform some task; for example, to display progress or save the current estimate. Then the application must re-enter `KTR_solve()` so that KNITRO can begin a new major iteration. `KTR_RC_NEWPOINT` is only returned if user option `KTR_PARAM_NEWPOINT=1`.

Section 4.2 describes example program that uses the reverse communications mode.

9.8 Callback mode for invoking KNITRO

The callback mode of KNITRO requires the user to supply several function pointers that KNITRO calls when it needs new function, gradient or Hessian values, or to execute a user-provided newpoint routine. For convenience, every one of these callback routines takes the same list of arguments. If your callback requires additional parameters, you are encouraged to create a structure containing them and pass its address as the `userParams` pointer. KNITRO does not modify or dereference the `userParams` pointer, so it is safe to use for this purpose. Section 4.2 describes an example program that uses the callback mode.

The C language prototype for the KNITRO callback function is defined in knitro.h:

```
typedef int KTR_callback (const int             evalRequestCode,
                          const int             n,
                          const int             m,
                          const int             nnzJ,
                          const int             nnzH,
                          const double * const  x,
                          const double * const  lambda,
                                double * const  obj,
                                double * const  c,
                                double * const  objGrad,
                                double * const  jac,
                                double * const  hessian,
                                double * const  hessVector,
                                void   *        userParams);
```

The callback functions for evaluating the functions, gradients and Hessian or for performing some newpoint task, are set as described below. Each user callback routine should return an `int` value of 0 if successful, or a negative value to indicate that an error occurred during execution of the user-provided function. Section 4.2 describes example program that uses the callback mode.

```
/* This callback should modify "obj" and "c". */
int KTR_set_func_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify "objGrad" and "jac". */
int KTR_set_grad_callback (KTR_context_ptr kc, KTR_callback * func);
```

```
/* This callback should modify "hessian" or "hessVector",
   depending on the value of "evalRequestCode". */
int KTR_set_hess_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify nothing. */
int KTR_set_newpoint_callback (KTR_context_ptr kc, KTR_callback * func);
```

NOTE: To enable ”newpoint” callbacks, set `KTR_PARAM_NEWPOINT=1`.

KNITRO also provides a special callback function for output printing. By default KNITRO prints to stdout or a `knitro.log` file, as determined by the `KTR_PARAM_OUTMODE` option. Alternatively, the user can define a callback function to handle all output. This callback function can be set as shown below.

```
int KTR_set_puts_callback (KTR_context_ptr kc, KTR_puts * puts_func);
```

The prototype for the KNITRO callback function used for handling output is:

```
typedef int KTR_puts (char * str, void * user);
```

10 Special problem classes

This section describes specializations in KNITRO to deal with particular classes of optimization problems. We also provide guidance on how to best set user options and model your problem to get the best performance out of KNITRO for particular types of problems.

10.1 Linear programming problems (LPs)

A linear program (LP) is an optimization problem where the objective function and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving LPs. However, KNITRO is unable to automatically detect whether or not a problem is an LP. In order for KNITRO to detect that a problem is an LP, you must specify this by setting the value of `objType` to `KTR_OBJTYPE_LINEAR` and all values of the array `cType` to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()` (see section 4). If this is not done, KNITRO will not apply special treatment to the LP and will typically be less efficient in solving the LP.

10.2 Quadratic programming problems (QPs)

A quadratic program (QP) is an optimization problem where the objective function is quadratic and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving QPs. However, KNITRO is unable to automatically detect whether or not a problem is a QP. In order for KNITRO to detect that a problem is a QP, you must specify this by setting the value of `objType` to `KTR_OBJTYPE_QUADRATIC` and all values of the array `cType` to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()` (see section 4). If this is not done, KNITRO will not apply special treatment to the QP and will typically be less efficient in solving the QP.

Typically, these specialization will only help on convex QPs.

10.3 Systems of Nonlinear Equations

KNITRO is quite effective at solving systems of nonlinear equations. To solve a square system of nonlinear equations using KNITRO one should specify the nonlinear equations as equality constraints (1.1*b*), and specify the objective function (1.1*a*) as zero (i.e., $f(x) = 0$).

10.4 Least Squares Problems

There are two ways of using KNITRO for solving problems in which the objective function is a sum of squares of the form

$$f(x) = \tfrac{1}{2} \sum_{j=1}^{q} r_j(x)^2.$$

If the value of the objective function at the solution is not close to zero (the large residual case), the least squares structure of f can be ignored and the problem can be solved as any other optimization problem. Any of the KNITRO options can be used.

On the other hand, if the optimal objective function value is expected to be small (small residual case) then KNITRO can implement the Gauss-Newton or Levenberg-Marquardt methods which only require first

derivatives of the residual functions, $r_j(x)$, and yet converge rapidly. To do so, the user need only define the Hessian of f to be

$$\nabla^2 f(x) = J(x)^T J(x),$$

where

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right] \begin{array}{l} j = 1,2,\ldots,q \\ i = 1,2,\ldots,n \end{array}.$$

The actual Hessian is given by

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{q} r_j(x)\nabla^2 r_j(x);$$

the Gauss-Newton and Levenberg-Marquardt approaches consist of ignoring the last term in the Hessian.

KNITRO will behave like a Gauss-Newton method by setting `KTR_PARAM_ALG=1`, and will be very similar to the classical Levenberg-Marquardt method when `KTR_PARAM_ALG=2`. For a discussion of these methods see, for example, [10].

10.5 Mathematical programs with equilibrium constraints (MPECs)

A mathematical program with equilibrium (or complementarity) constraints (also know as an MPEC or MPCC) is an optimization problem which contains a particular type of constraint referred to as a complementarity constraint. A complementarity constraint is a constraint which enforces that two variables are *complementary* to each other, i.e., the variables x_1 and x_2 are complementary if the following conditions hold

$$x_1 \times x_2 = 0, \quad x_1 \geq 0, \quad x_2 \geq 0. \tag{10.27}$$

The condition above, is sometimes expressed more compactly as

$$0 \leq x_1 \quad \perp \quad x_2 \geq 0.$$

One could also have more generally, that a particular constraint is complementary to another constraint or a constraint is complementary to a variable. However, by adding slack variables, a complementarity constraint can always be expressed as two variables complementary to each other, and KNITRO requires that you express complementarity constraints in this form. For example, if you have two constraints $c_1(x)$ and $c_2(x)$ which are complementary

$$c_1(x) \times c_2(x) = 0, \quad c_1(x) \geq 0, \quad c_2(x) \geq 0,$$

you can re-write this as two equality constraints and two complementary variables, s_1 and s_2 as follows:

$$s_1 = c_1(x) \tag{10.28}$$
$$s_2 = c_2(x) \tag{10.29}$$
$$s_1 \times s_2 = 0, \quad s_1 \geq 0, \quad s_2 \geq 0. \tag{10.30}$$

Intuitively, a complementarity constraint is a way to model a constraint which is combinatorial in nature since, for example, the conditions in (10.27) imply that either x_1 or x_2 must be 0 (both may be 0 as well). Without special care, these type of constraints may cause problems for nonlinear optimization solvers because problems which contain these types of constraints fail to satisfy constraint qualifications which are often

assumed in the theory and design of algorithms for nonlinear optimization. For this, reason we provide a special interface in KNITRO for specifying complementarity constraints. In this way, KNITRO can recognize these constraints and apply some special care to them internally.

Complementarity constraints can be specified in KNITRO through a call to the function `KTR_addcompcons()` which has the following prototype and argument list.

Prototype:

```
int KTR_addcompcons(KTR_context_ptr kc,
                    int numCompConstraints,
                    int * indexList1,
                    int * indexList2);
```

Arguments:

`KTR_context *kc:` is a pointer to a structure which holds all the relevant information about a particular problem instance.

`int numCompConstraints:` is a scalar specifying the number of complementarity constraints to be added to the problem (i.e., the number of pairs of variables which are complementary to each other).

`int *indexList1:` is an array of length `numCompConstraints` specifying the variable indices for the first set of variables in the pairs of complementary variables.

`int *indexList2:` is an array of length `numCompConstraints` specifying the variable indices for the second set of variables in the pairs of complementary variables.

The call to `KTR_addcompcons()` must occur after the call to `KTR_init_problem()`, but before the first call to `KTR_solve()`. Below we provide a simple example of how to define the KNITRO data structures to specify a problem which includes complementarity constraints.

Example: [1]

Assume we want to solve the following MPEC with KNITRO.

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0-5)^2 + (2x_1+1)^2 && \text{(10.31a)}\\
\text{subject to} \quad & c_0(x) = 2(x_1-1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 && \text{(10.31b)}\\
& c_1(x) = 3x_0 - x_1 - 3 \geq 0 && \text{(10.31c)}\\
& c_2(x) = -x_0 + 0.5x_1 + 4 \geq 0 && \text{(10.31d)}\\
& c_3(x) = -x_0 - x_1 + 7 \geq 0 && \text{(10.31e)}\\
& x_i \geq 0, i = 0..4 && \text{(10.31f)}\\
& c_1(x)x_2 = 0 && \text{(10.31g)}\\
& c_2(x)x_3 = 0 && \text{(10.31h)}\\
& c_3(x)x_4 = 0. && \text{(10.31i)}
\end{aligned}$$

[1] An MPEC from J.F. Bard, Convex two-level optimization, *Mathematical Programming* 40(1), 15-27, 1988.

It is easy to see that the last 3 constraints (along with the corresponding non-negativity conditions) represent complementarity constraints. Expressing this in compact notation, we have:

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0-5)^2 + (2x_1+1)^2 && \text{(10.32a)}\\
\text{subject to} \quad & c_0(x) = 0 && \text{(10.32b)}\\
& 0 \le c_1(x) \perp x_2 \ge 0 && \text{(10.32c)}\\
& 0 \le c_2(x) \perp x_3 \ge 0 && \text{(10.32d)}\\
& 0 \le c_3(x) \perp x_4 \ge 0 && \text{(10.32e)}\\
& x_0 \ge 0, x_1 \ge 0. && \text{(10.32f)}
\end{aligned}$$

Since KNITRO requires that complementarity constraints be written as two variables complementary to each other, we must introduce slack variables x_5, x_6, x_7 and re-write problem (10.31) as

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0-5)^2 + (2x_1+1)^2 && \text{(10.33a)}\\
\text{subject to} \quad & c_0(x) = 2(x_1-1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 && \text{(10.33b)}\\
& \tilde{c_1}(x) = 3x_0 - x_1 - 3 - x_5 = 0 && \text{(10.33c)}\\
& \tilde{c_2}(x) = -x_0 + 0.5x_1 + 4 - x_6 = 0 && \text{(10.33d)}\\
& \tilde{c_3}(x) = -x_0 - x_1 + 7 - x_7 = 0 && \text{(10.33e)}\\
& x_i \ge 0, i = 0..7 && \text{(10.33f)}\\
& x_2 \perp x_5 && \text{(10.33g)}\\
& x_3 \perp x_6 && \text{(10.33h)}\\
& x_4 \perp x_7. && \text{(10.33i)}
\end{aligned}$$

Now that the problem is in a form suitable for KNITRO, we define the problem for KNITRO by using `c`, `cLoBnds`, and `cUpBnds` for (10.33*b*)-(10.33*e*), and `xLoBnds`, `xUpBnds` for (10.33*f*) to specify the normal constraints and bounds in the usual way for KNITRO. We use `indexList1`, `indexList2` and the `KTR_addcompcons()` function call to specify the complementarity constraints (10.33*g*)-(10.33*i*). These arrays are specified as follows for (10.33).

```
n = 8;                  /* number of variables */
m = 4;                  /* number of regular constraints */
numCompConstraints = 3; /* number of complementarity constraints */

c[0] = 2*(x[1]-1) - 1.5*x[0] + x[2] - 0.5*x[3] + x[4];
c[1] = 3*x[0] - x[1] - 3 -x[5];
c[2] = -x[0] + 0.5*x[1] + 4 -x[6];
c[3] = -x[0] - x[1] + 7 - x[7];

cLoBnds[0] = 0;  cUpBnds[0] = 0;
cLoBnds[1] = 0;  cUpBnds[1] = 0;
cLoBnds[2] = 0;  cUpBnds[2] = 0;
cLoBnds[3] = 0;  cUpBnds[3] = 0;
```

```
xLoBnds[0] = 0;  xUpBnds[0] = KTR_INFBOUND;
xLoBnds[1] = 0;  xUpBnds[1] = KTR_INFBOUND;
xLoBnds[2] = 0;  xUpBnds[2] = KTR_INFBOUND;
xLoBnds[3] = 0;  xUpBnds[3] = KTR_INFBOUND;
xLoBnds[4] = 0;  xUpBnds[4] = KTR_INFBOUND;
xLoBnds[5] = 0;  xUpBnds[5] = KTR_INFBOUND;
xLoBnds[6] = 0;  xUpBnds[6] = KTR_INFBOUND;
xLoBnds[7] = 0;  xUpBnds[7] = KTR_INFBOUND;

indexList1[0] = 2;   indexList2[0] = 5;
indexList1[1] = 3;   indexList2[1] = 6;
indexList1[2] = 4;   indexList2[2] = 7;
```

NOTE: Variables which are specified as complementary through the special `KTR_addcompcons()` functions should be specified to have a lower bound of 0 through the KNITRO lower bound array `xLoBnds`.

When using KNITRO through a particular modeling language, only some modeling languages allow for the identification of complementarity constraints. If a modeling language does not allow you to specifically identify and express complementarity constraints, then these constraints must be formulated as regular constraints and KNITRO will not perform any specializations.

10.6 Global optimization

KNITRO is designed for finding locally optimal solutions of continuous optimization problems. A local solution is a feasible point at which the objective function value at that point is as good or better than at any "nearby" feasible point. A globally optimal solution is one which gives the best (i.e., lowest if minimizing) value of the objective function out of all feasible points. If the problem is *convex* all locally optimal solutions are also globally optimal solutions. The ability to guarantee convergence to the global solution on large-scale *nonconvex* problems is a nearly impossible task on most problems unless the problem has some special structure or the person modeling the problem has some special knowledge about the geometry of the problem. Even finding local solutions to large-scale, nonlinear, nonconvex problems is quite challenging.

Although KNITRO is unable to guarantee convergence to global solutions it does provide a *multi-start* heuristic which attempts to find multiple local solutions in the hopes of locating the global solution. See section 9.6 for information on trying to find the globally optimal solution using the KNITRO multi-start feature.

References

[1] R. H. Byrd, J.-Ch. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.

[2] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz. On the convergence of successive linear-quadratic programming algorithms. Technical Report OTC 2002/5, Optimization Technology Center, Northwestern University, Evanston, IL, USA, 2002. To appear in SIAM Journal on Optimization.

[3] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz. An algorithm for nonlinear optimization using linear programming and equality constrained subproblems. *Mathematical Programming, Series B*, 100(1):27–48, 2004.

[4] R. H. Byrd, M. E. Hribar, and J. Nocedal. An interior point algorithm for large scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900, 1999.

[5] R. H. Byrd, J. Nocedal, and R. A. Waltz. Feasible interior methods using slacks for nonlinear optimization. *Computational Optimization and Applications*, 26(1):35–61, 2003.

[6] R. H. Byrd, J. Nocedal, and R.A. Waltz. KNITRO: An integrated package for nonlinear optimization. In G. di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pages 35–59. Springer, 2006.

[7] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, 1993.

[8] Harwell Subroutine Library. *A catalogue of subroutines (HSL 2002)*. AEA Technology, Harwell, Oxfordshire, England, 2002.

[9] Hock, W. and Schittkowski, K. *Test Examples for Nonlinear Programming Codes*, volume 187 of *Lecture Notes in Economics and Mathematical Systems*. Springer-Verlag, 1981.

[10] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 1999.

[11] R. A. Waltz, J. L. Morales, J. Nocedal, and D. Orban. An interior algorithm for nonlinear optimization that combines line search and trust region steps. Technical Report 2003-6, Optimization Technology Center, Northwestern University, Evanston, IL, USA, June 2003. To appear in Mathematical Programming A.

Appendix A

Solution Status Codes

0: `EXIT: LOCALLY OPTIMAL SOLUTION FOUND.`

KNITRO found a locally optimal point which satisfies the stopping criterion (see section 6 for more detail on how this is defined). If the problem is convex (for example, a linear program), then this point corresponds to a globally optimal solution.

-1: `EXIT: Iteration limit reached.`

The iteration limit was reached before being able to satisfy the required stopping criteria.

-2: `EXIT: Convergence to an infeasible point.`
`Problem may be locally infeasible.`

The algorithm has converged to an infeasible point from which it cannot further decrease the infeasibility measure. This happens when the problem is infeasible, but may also occur on occasion for feasible problems with nonlinear constraints or badly scaled problems. It is recommended to try various initial points. If this occurs for a variety of initial points, it is likely the problem is infeasible.

-3: `EXIT: Problem appears to be unbounded.`
`Iterate is feasible and objective magnitude > objrange.`

The objective function appears to be decreasing without bound, while satisfying the constraints. If the problem really is bounded, increase the size of the parameter `objrange` to avoid terminating with this message.

-4: `EXIT: Relative change in solution estimate < xtol.`

The relative change in the solution estimate is less than that specified by the paramater `xtol`. To try to get more accuracy one may decrease `xtol`. If `xtol` is very small already, it is an indication that no more significant progress can be made. If the current point is feasible, it is possible it may be optimal, however the stopping tests cannot be satisfied (perhaps because of degeneracy, ill-conditioning or bad scaling).

-5: `EXIT: Current solution estimate cannot be improved. Point appears to be optimal, but desired accuracy could not be achieved.`

No more progress can be made, but the stopping tests are close to being satisfied (within a factor of 100) and so the current approximate solution is believed to be optimal.

-6: `EXIT: Time limit reached.`

The time limit was reached before being able to satisfy the required stopping criteria.

-50 to -60:

Termination values in this range imply some input error. If `outlev>0` details of this error will be printed to standard output or the file knitro.log depending on the value of `outmode`.

-90: `EXIT: Callback function error.`

This termination value indicates that an error (i.e., negative return value) occurred in a user provided callback routine.

-97: `EXIT: LP solver error.`

This termination value indicates that an unrecoverable error occurred in the LP solver used in the active-set algorithm preventing the optimization from continuing.

-98: `EXIT: Evaluation error.`

This termination value indicates that an evaluation error occurred (e.g., divide by 0, taking the square root of a negative number), preventing the optimization from continuing.

-99: `EXIT: Not enough memory available to solve problem.`

This termination value indicates that there was not enough memory available to solve the problem.

Appendix B

Migrating to KNITRO 5.0

Migrating to KNITRO 5.0

KNITRO 5.0 is NOT backwards compatible with previous versions of KNITRO. However, it should be a simple process to migrate from KNITRO 4.0 to 5.0 as the primary data structures have not changed. Whereas KNITRO 4.0 solved problems through a sequence of three function calls:

- `KTR_new()`
- `KTR_solve()`
- `KTR_free()`

KNITRO 5.0 uses a sequence of four function calls:

- `KTR_new()`
- `KTR_init_problem()`
- `KTR_solve()`
- `KTR_free()`

Here, `KTR_init_problem()` is a new function call used to pass in the optimization problem definition. There is also a new function call `KTR_restart()` for re-solving the same problem with a different initial point or different user option settings.

Summary of API changes:

- The argument list in the function call `KTR_new()` has changed. No arguments are passed.
- A new function `KTR_init_problem()` was created to pass information which describes the structure of your problem.
- The argument list in the function call `KTR_solve()` has changed. Many variables which used to be passed through `KTR_solve()` are now passed through `KTR_init_problem()`.
- A new argument `objGoal` was created to specify whether the problem is formulated as a minimization problem or a maximization problem. This argument is passed through `KTR_init_problem()`.
- Many variable names have changed to be more descriptive (although their structure is the same).
- New functions of the form

  `KTR_get_*`

  were created for retrieving solution information (see section 7).

See section 4 for detailed information on using the KNITRO 5.0 API. In addition numerous sample programs are provided in the examples directory of the distribution.